Guide to ThemWi RTP endpoint library

Mychaela N. Falconia

Themyscira Wireless

Version 2.1, last edited 2025/08/21

1. Introduction

There exists an RTP endpoint implementation that was developed by the present author under Themyscira Wireless branding, for use in application programs that are built with the `libosmo*` suite of Osmocom libraries. This RTP implementation is called **twrtp**, short for Themyscira Wireless RTP endpoint library, and exists in several versions:

1. The original version resides in a git repository named `twrtp-proto` in ThemWi area of Osmocom Gitea. As reflected in its name, this version reflects a prototype stage in development: later-invented features are not present, and the API is not polished. This `twrtp-proto` library was used in early versions of `tw-border-mgw`, which proved **twrtp** as production-worthy through use on the author's depended-upon home GSM network.
2. The new official, Themyscira-endorsed version of **twrtp** resides in the git repository named `twrtp-native`, also hosted in ThemWi area of Osmocom Gitea. This version is not API-compatible with `twrtp-proto`; instead it builds on the lessons learned from the former. This version of **twrtp** shall be used in updated versions of `tw-border-mgw` and `tw-e1abis-mgw`, and in all new Themyscira software going forward.

   All header file names and C API symbol prefixes are in ThemWi namespace in this version of **twrtp**, not in Osmocom namespace.
3. At the time of this writing, an effort is underway to get a version of **twrtp** merged into mainline `libosmo-netif`; WIP patches are in Osmocom Gerrit. The purpose of this initiative is to make it possible to use **twrtp** instead of Belledonne **ortp** library (as an option) in OsmoBTS, thereby eliminating Belledonne software from the production code path in GSM networks that are built from Osmocom and ThemWi components according to ThemWi architectural principles.

   Feedback received from Osmocom maintainers indicates that the desire to replace **ortp** in OsmoBTS with a new library that is built on top of `libosmocore` primitives (such as **twrtp**) exists beyond ThemWi, hence the idea of **twrtp** is generally welcome. However, because non-ThemWi members of Osmocom community desire to use the proposed RTP endpoint library in environments and applications that can be significantly different from Themyscira environment of TDM emulation, the derivative of **twrtp** that is expected to be merged into `libosmo-netif` will likely contain modifications such that it will no longer be pure ThemWi algorithms.

The principal version of **twrtp** described in this document is `twrtp-native`, along with some mention of `twrtp-proto`: these two versions are the only ones over which this author retains sole authority. The alternate version submitted as a candidate for inclusion in Osmocom mainline will have its own documentation submitted as part of the patch series, so it can be updated in sync with Osmocom-driven changes to algorithms and APIs in that version.

1.1. Principal function

ThemWi RTP endpoint library, consisting of `<themwi/rtp/endp.h>` and `<themwi/rtp/twjit.h>` blocks in `twrtp-proto` or `<themwi/rtp2/twrtp.h>` and `<themwi/rtp2/twjit.h>` blocks in `twrtp-native`, is intended for interworking between an RTP stream and a fixed timing system such as GSM Um interface TCH or T1/E1 TDM. Such interworking consists of two fundamental elements:

- In every fixed time quantum, the interworking element receives a unit of speech or CSData media from the fixed source and emits an RTP packet carrying that quantum of speech or CSData.
- In the opposite direction, the fixed timing system requires a quantum of speech or CSData to be fed to it on every tick without fail, yet the interworking element has no control over when RTP packets may arrive from the IP network. This direction of interworking requires a rather complex element called a jitter buffer, an element whose design and configuration always involves some trade-offs and compromises.

1.2. Domain of application

The present library is **not** intended to be an all-purpose implementation of IETF RFCs 3550 and 3551, supporting all possible RTP use cases as envisioned by IETF. Instead it is intended to support RTP *as it is used* in these two specific telecom network environments:

- 3GPP networks that use RTP according to TS 26.102 and TS 48.103;
- The way RTP is used to transport G.711 PSTN traffic across the public Internet in what may be colloquially referred to as IP-PSTN.

The two 3GPP specs referenced above prescribe a fixed packetization time of 20 ms for all codecs on AoIP interface. Furthermore, they stipulate that:

- In the case of compressed speech transport, each RTP packet carries exactly one frame of the speech codec in use;
- In the case of uncompressed G.711 speech or CSData transport, each RTP packet carries exactly 160 payload octets (20 ms worth) of what would have been a 64 kbit/s timeslot in T1/E1 transport.

This fixed-quantum property, namely the property that every RTP packet carries exactly one fixed quantum of speech or CSData, where the duration of this quantum is known at connection setup time and cannot suddenly change from one packet to the next, is required by the present ThemWi RTP endpoint library — this requirement constitutes a fundamental aspect of its architectural design.

An RTP endpoint implementation library that imposes the just-described requirement is sufficient for the purpose of building IP-based GSM networks that follow 3GPP TS 48.103 (the requirements of that spec are in agreement with the library constraint), and it is also sufficient for interfacing to IP-PSTN by way of common commercial PSTN-via-SIP connectivity providers.

In the case of IP-PSTN, the author of the present library has experience only with North American PSTN-via-SIP connectivity providers. In all of our operational experience so far, these IP-PSTN connectivity providers behave in ways that are fully compatible with the expectations of the present RTP library, as long as the following conditions are met:

- No attempt is made to use any codecs other than PCMU or PCMA: don't include any other codecs in the SDP offer, and send only SDP answers that select either PCMU or PCMA out of the received multi-codec offer.
- No attempt is made to use any other `ptime` besides the most common industry standard of 20 ms.

In all operational experience so far, incoming INVITE SDPs indicate either `a=ptime:20` or `a=maxptime:20`, and when we indicate `a=ptime:20` in all SDPs we send out, the IP-PSTN peer always sends us 20 ms RTP packets, as opposed to some other packetization interval which would break the fixed-quantum model assumed by the present RTP library.

However, it needs to be acknowledged that the present library is **not** suitable for general-purpose, IETF-style applications outside of ‘‘walled garden’’ 3GPP networks or the semi-walled environment of IP-PSTN with ‘‘well-behaved’’ entities: there are many behaviors that are perfectly legal per the RFCs, but are not supported by the present library. Having a peer send RTP with a packetization interval that is different from what we asked for via `ptime` attribute is one of those behaviors that is allowed by IETF, but not supported by this library.

1.2.1. Expectation of continuous streaming

In addition to the just-described requirement for a fixed packetization interval, the domain of application for **twrtp** is subject to one more constraint: our jitter buffer component (**twjit**) is designed for environments that implement continuous streaming, and may perform suboptimally in those that do not.

Continuous streaming is an operational policy under which an RTP endpoint *always* emits an RTP packet in *every* 20 ms (or whatever other packetization interval is used) time window, be it rain or shine, even if it has no data to send because nothing was received on the air interface (DTX pause on the radio link, reception errors, frame stealing) or because E1 TRAU frame decoding failed, etc. Continuous streaming may be implemented by sending an RTP packet with a zero-length payload when the endpoint has nothing else to send in a given quantum time window — this method allows any existing RTP payload format standard to be operationally modified for continuous streaming. There also exist Themyscira-defined enhanced RTP payload formats for GSM speech codecs that not only mandate continuous streaming, but additionally convey errored frame bits and the Time Alignment Flag in every 20 ms frame position, exactly like TRAU-UL frames in the world of TDM-based GSM.

The opposite of continuous streaming is the practice of intentional gaps. Under this operational policy, an RTP endpoint may create intentional gaps in the RTP stream it emits, simply by sending no RTP packets at all when it deems that there are no useful data to be transmitted. An intentional gap is distinguished from packet loss in that the sequence number in the RTP header increments by one while the timestamp increments by a greater than normal amount. Unfortunately for **twjit**, intentional gaps in RTP were the design intent of IETF. Even more unfortunately, this IETF-ism has been canonized by 3GPP in TS 26.102 and TS 48.103 — hence those operators who prefer a continuous streaming model now have to explicitly deviate from 3GPP specifications.

In an Osmocom GSM network, 3GPP-compliant operation with intentional gaps is the default — however, the operator can switch to continuous streaming model by setting `rtp continuous-streaming` in OsmoBTS vty configuration.

Fortunately for **twjit**, however, the situation is better in the world of IP-PSTN, the other RTP environment for which the present library was designed. At least on North American IP-PSTN and at least when uncompressed PCMU or PCMA codecs are used, all PSTN-via-SIP connectivity providers in our operational experience so far always emit perfectly continuous RTP streams, without any intentional gaps.

If an application uses **twrtp** with **twjit** to receive an RTP stream that incurs intentional gaps, the resulting performance may be acceptable or unacceptable depending on additional factors:

- If RTP gaps are incurred only during frame erasure events (radio errors or FACCH stealing) without DTX, the resulting **twjit** performance will most likely still be acceptable for speech applications. All transmitted speech frames will still be delivered to the receiver, but the frame erasure gap may lengthen or shorten depending on exact jitter buffer conditions at the time of the intentional gap in the Tx stream — in other words, a phase shift may be incurred.
- If RTP stream gaps are enabled in conjunction with DTX, **twjit** will not be able to receive such a stream according to common expectations. When RTP stream gaps are used together with DTX, the stream will typically feature occasional single packets of comfort noise update, sent every 160, 240 or 480 ms depending on the codec, surrounded by gaps. When **twjit** receives such a stream and the flow-starting fill level is set to 2 or greater (the default and usually necessary configuration), all of these ‘‘isolated island’’ comfort noise update packets will be dropped — a behavior counter to the way DTX is expected to work.

The take-away is that if an operator wishes to use DTX with **twrtp**, they need to enable `rtp continuous-streaming`.

1.3. Configurable quantum duration and time scale

For every RTP stream it handles, the library needs to know two key parameters:

- The scale or ‘‘clock rate’’ used for RTP timestamps, i.e., how many timestamp units equal one millisecond of physical time;

- The ‘‘quantum’’ duration in milliseconds.

  Quantum is the term used in this RTP endpoint library for the unit of speech or CSData carried in one RTP packet. In Kantian philosophy terms, a quantum of speech or CSData is the thing-in-itself (a single codec frame, or a contiguous chunk of 160 PCM samples grabbed from an ISDN B channel), whereas the RTP packet that carries said quantum is one particular transport representation of that thing-in-itself.

  In most applications of this library (all 3GPP codecs other than AMR-WB, and all IP-PSTN applications in our experience so far), the time scale is 8000 timestamp units per second (or 8 per millisecond, as it appears in the actual APIs) and the time duration of a single quantum is 20 ms, hence one quantum equals 160 timestamp units. Both parameters (RTP timestamp clock rate in kHz and the number of ms per quantum) are configurable at the time of endpoint creation, allowing RTP endpoints for AMR-WB, or perhaps G.711 or CSData applications with different packetization times — but they cannot be changed later in the lifetime of an allocated endpoint.

  For ease of exposition, the rest of this document will assume that one quantum equals 20 ms in time or 160 RTP timestamp units. If these numbers are different in your application, substitute accordingly.

2. Jitter buffer model

In the interworking direction from incoming RTP to the fixed timing system, the latter will poll the RTP endpoint (or more precisely, the jitter buffer portion thereof) for a quantum of media every 20 ms, whenever that quantum is required for transmission on GSM Um TCH or for TDM output etc. The job of the jitter buffer is to match previously received RTP packets to these fixed-timing output polls, while striving to meet these two conflicting goals:

- Any time that elapses between an RTP packet being received and its payload being passed as a quantum to the fixed timing system constitutes added latency — which needs to be minimized.
- IP-based transport always involves some jitter: the time delta between the receipt of one RTP packet and the arrival of its successor is very unlikely to be exactly equal to 20 ms every time. This jitter may be already present in the RTP stream from its source if that source is an IP-native BTS that does not pass through E1 Abis and thus exposes the inherent jitter of GSM TDMA multiframe structure, but even if the source is perfectly timed, some jitter will still be seen on the receiving end. Depending on the actual amount of jitter seen in a given deployment, it may be necessary to introduce some latency-adding buffering in the receiving RTP endpoint — otherwise the function of interworking to the fixed timing system at the destination will perform poorly, as will be seen in the ensuing sections.

This chapter covers the design and operation of **twjit**, the jitter buffer component of ThemWi RTP endpoint library.

2.1. Flows and handovers

In **twjit** terminology, a single RTP flow is a portion (or the whole) of an RTP stream that exhibits the following two key properties:

- All packets have the same SSRC;
- The RTP timestamp increment from each packet to the next always equals the fixed quantum duration expressed in timestamp units, i.e., 160 in most practical applications.

A handover in **twjit** terminology is a point in the incoming RTP stream at which either of the following events occurs:

- An SSRC change is seen;
- The RTP timestamp advances by an increment that is not an integral multiple of the expected fixed quantum duration. In contrast, if an RTP timestamp increment is seen that *is* an integral multiple of the quantum, but that integral multiple is more than one, and there is enough buffering in the system such that this event is seen before the jitter buffer underruns, such events are **not** treated as handovers: instead it is assumed to be an occurrence of either packet loss or reordering.

A handover in **twjit** is thus a transition from one flow to the next. This term was adopted because such transitions are expected to occur when an RTP stream belonging to a single call switches from one BSS endpoint to another (in the same BSS or in a different one) upon radio handover events in GSM and other cellular networks, but handovers in **twjit** sense can also occur in other applications that aren't GSM. For example, if an IP-PSTN peer we are conversing with suddenly decides, for its own reasons known only to itself, to change its SSRC or jump its RTP output timescale, our **twjit** instance will treat that event as a handover.

2.2. Examples of flows

The following drawing depicts the best case scenario of a TDM-native speech or CSData stream being transported across an IP network in RTP:

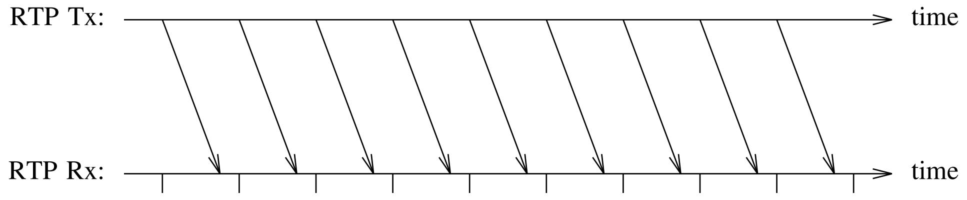


Figure 1: Ideal case

In the above figure and other similar drawings that follow, each down-and-forward arrow represents an RTP packet: the beginning of each arrow on the RTP Tx line is the point in time when that RTP packet is emitted by the source endpoint, and the landing point of each arrow on the RTP Rx line is the time point when the same packet is received at the destination endpoint. The forward horizontal movement of each arrow in the figure is the flight time of the corresponding RTP packet through the IP network. Tick marks below the RTP Rx time axis represent fixed points in time when the destination application polls its **twjit** buffer because the destination fixed timing system (GSM Um TCH, T1/E1 etc) requires a new quantum of media.

Figure 1 depicts the ideal scenario: the source endpoint emits RTP packets in a perfect 20 ms cadence without built-in jitter, the flight time through the IP network also remains constant from each packet to the next (no jitter introduced by IP transport), and these packets arrive in a perfect cadence at the receiving endpoint, exactly one packet before each **twjit** polling instant. Let us now consider a more realistic scenario:

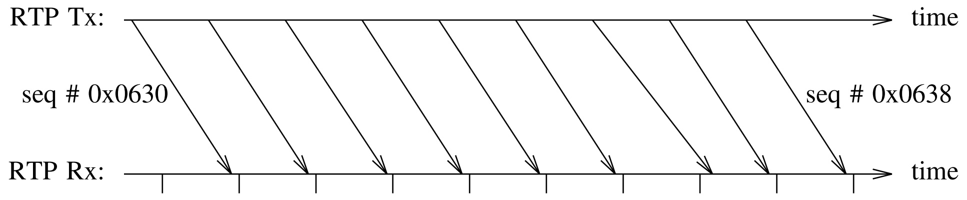


Figure 2: IP-PSTN realistic scenario, good Internet connection

Figure 2 above is based on an actual IP-PSTN test call that was made on 2024-05-14 from the author's interconnection point with BulkVS to a phone number on T-Mobile USA (2G). The figure was drawn using these time-of-arrival delta numbers from the pcap of that call:

From seq #	To seq #	ToA delta (ms)
0x0630	0x0631	19.992
0x0631	0x0632	20.522
0x0632	0x0633	19.509
0x0633	0x0634	20.211
0x0634	0x0635	19.741
0x0635	0x0636	25.245
0x0636	0x0637	14.776
0x0637	0x0638	20.007

This small excerpt from the pcap of one particular test call is a representative example of this author's general experience with North American IP-PSTN. Most of the time the Δ between arrival times of two successive RTP packets is within a few microseconds of the ideal 20 ms value; interarrival jitter spikes up to about 500 µs are fairly frequent (seen a few times every second), but larger spikes (in the range of several milliseconds) appear as rare outliers when I look at pcap files from short test calls.

In order to draw packet flight diagrams that are intuitively understandable by a human reader, we need to know the absolute flight time from the source for each depicted packet. In actual operation this absolute flight time is unknowable — the only available info is the observed cadence of time-of-arrival deltas. (The absolute reception time of each packet according to the local clock is known of course, but it is of no use without knowing the absolute time at which those packets were emitted by the sender.) These absolute flight times of packets are furthermore not needed for actual operation of jitter buffers — the available ToA Δ information is sufficient for jitter buffer design and tuning — but they are needed for more intuitive understanding by humans. Therefore, when we draw packet flight diagrams, we have to factor in some arbitrary, made-up number for the ''baseline'' packet flight time under ideal conditions: the purely notional ''baseline'' number which, when added to the actually observed jitter, equals what we assume to be the true flight time of each individual packet. In drawing Figure 2 above, I set this ''baseline'' delay to 26 ms: one half of the lowest round-trip time I observe now when I ping the IPv4 address of the IP-PSTN node I was conversing with in that test call.

In drawing this figure, I also exercised a degree of freedom in choosing the arbitrary (not known in advance in real operation) phase shift between the arrival time of RTP packets and the receiving entity's fixed time base, i.e., the position of fixed-time polling ticks below the RTP Rx time axis relative to the times of packet arrival on the same axis. The specific phase shift I chose in drawing this figure is one that illustrates the effect of this amount of real-world jitter on **twjit** operation: RTP packet with sequence number 0x0636 arrives just after the receiver's polling time instead of just before. The significance of this effect will be seen when we examine **twjit** operation and tuning.

At this point a reader of this paper, seeing that most packets depicted in Figure 2 exhibit interarrival jitter measured in µs rather than ms, with a single occurrence of 5 ms jitter as a rare outlier, may accuse this author of first-world privilege in terms of Internet connection quality. So let us consider what the figure would look like with more substantial jitter — but still below 20 ms. (Why below 20 ms, you may ask? The answer will be seen shortly.) Because such jitter does not occur in the wild where I live, I used a made-up dataset for the following figure:

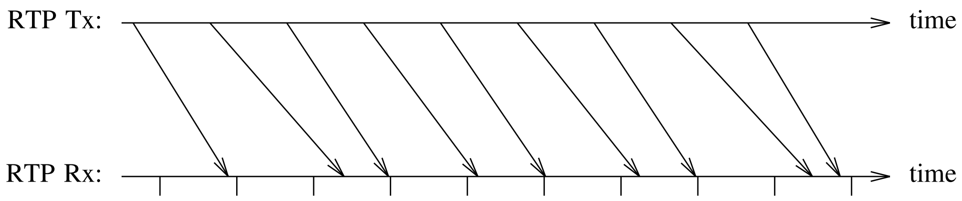


Figure 3: 13 ms of random jitter

In the above figure, each packet flight time was arbitrarily picked in the range between 24 and 37 ms, i.e., a ''baseline'' delay of 24 ms combined with 13 ms of jitter. (The actual flight times for the figure were initially drawn from an RNG program, then slightly tweaked by hand to get closer to the extremes of the jitter range to be illustrated.) Unlike Figure 2 which represents a real life occurrence, Figure 3 is completely made up — yet as will be seen later in this paper, the same **twjit** configuration that is optimal for Figure 2 will also handle the conditions of Figure 3 just as well — and the jitter is more visible here.

So far we've only considered cases of jitter below 20 ms, i.e., jitter magnitude less than the periodic interval between successive RTP packets. A reader ought to ask now: what happens if the jitter exceeds 20 ms? Before we can answer the question of what happens in such cases, let us first consider what it means for IP network-induced jitter to exceed the interval between successive packets. Assuming that successive RTP packets are emitted every 20 ms by the sender, for the receiver to experience interarrival jitter that exceeds 20 ms, the intervening IP network would have to ''bunch up'' packets: the receiving end suddenly stops receiving packets when they are expected, then a slew of massively delayed packets arrive all at once. The following figure is a real life example of such occurrence:

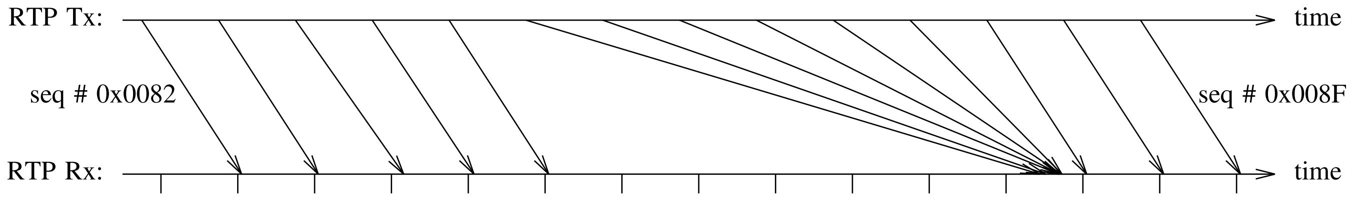


Figure 4: 6 packets bunched together by IP network bottleneck

The above figure is based on observed behavior during an experiment performed by this author on 2024-03-31, involving a mobile Internet connection (LTE) and a Hurricane Electric IPv6 tunnel. A WireGuard tunnel was established between the author's laptop and a server; the test laptop was connected to the Internet via T-Mobile LTE in this experiment, while the server was one that has native IPv4 plus an IPv6 address by way of HE 6-in-4 tunnel. The WireGuard tunnel was set up using only IPv6 addresses on the outside, i.e., the LTE leg saw only IPv6 in this experiment. A test stream of RTP packets, spaced 20 ms apart on the sending end, was transmitted inside a WireGuard tunnel from the test laptop to the test server; the path of each packet was thus as follows:

- WireGuard encapsulation in IPv6 on the sending end (laptop);
- Transport across T-Mobile Internet service (LTE) in IPv6, going to the IPv6 address of the Hurricane Electric tunnel;
- Hurricane Electric PoP received each packet on IPv6 and re-emitted it in IPv4 wrapping, going to the IPv4 address of the author's server;
- The receiving server decapsulated first 6-in-4, then WireGuard.

A similar experiment was performed addressing a different server, one that has native IPv6 connectivity in addition to IPv4; the behavior seen in Figure 4 was not seen in that other experiment, leading to the conclusion that the IP network bottleneck that occasionally ''bunches together'' a series of consecutive RTP packets is an artifact of the HE 6-in-4 tunnel, rather than an artifact of mobile Internet access via LTE.

Irrespective of the cause though, Figure 4 is a good illustration of what happens when buffering delays at an IP network bottleneck significantly exceed the spacing interval between successive RTP packets. No packet loss occurred in this experiment, i.e., every packet emitted by the sending end was *eventually* received; likewise, no reordering appeared at the receiving end: packets were received in the same order in which they were emitted by the sender. However, if we look at the arrival times of these selected packets on the receiving end, we see the following picture — the dataset from which Figure 4 was drawn:

From seq #	To seq #	ToA delta (ms)
0x0082	0x0083	19.951
0x0083	0x0084	22.235
0x0084	0x0085	18.341
0x0085	0x0086	19.429
0x0086	0x0087	127.784
0x0087	0x0088	1.542
0x0088	0x0089	3.417
0x0089	0x008A	0.273
0x008A	0x008B	0.440
0x008B	0x008C	0.115
0x008C	0x008D	6.467
0x008D	0x008E	20.013
0x008E	0x008F	20.009

Generally speaking, IP network behavior in this more adverse environment (passing through a leg of consumer mobile Internet) is not much worse than the ''luxurious'' IP-PSTN environment (server to server, business-grade Internet connection on the non-datacenter end) of Figure 2: most of the time, ToA Δ from one packet to the next is only a few μs away from the true 20 ms ideal, with occasional jitter spikes of a few ms. However, occasionally a more obstinent bottleneck occurs in the IP network path that blocks the flow for a much longer duration: in the example I presented here, for just over 120 ms. During such suddenly induced pauses, RTP packets coming from the source every 20 ms accumulate at the bottleneck, and when that bottleneck clears, all queued-up packets are delivered directly back to back, arriving less than 1 ms apart, essentially all at once.

In all examples we have considered so far, there has been no packet reordering: despite variations in flight time that appear on the receiving end as jitter, all RTP packets were received in the same order in which they were emitted by the source endpoint. Now let us consider what kind of scenarios can result in RTP packets arriving out of order. So far this author has not observed even one actual occurrence of packet reordering — apparently it does not happen on IP networks that exist in this part of the world, even on consumer LTE — hence the following analysis will be strictly theoretical. Given that the source endpoint steadily emits one packet at a time, spaced every 20 ms, how can these packets arrive in a reversed order? In order for packets to arrive out of order, a later-sent packet has to experience significantly shorter transit delay than an earlier-sent one, such that the later-sent packet ‘‘overtakes’’ the earlier-sent one. One situation where we can easily imagine such happening is the ‘‘melee’’ shown in Figure 4, the spot on the RTP Rx time axis where 6 different arrows, representing 6 different RTP packets emitted 20 ms apart, all arrive at essentially the same point in time. In our actual experience, such ‘‘bunched together’’ packets still arrive in the correct order, even if the Δ in the time of arrival between them is only a few μs — but we can easily imagine a different implementation of the offending IP network element (the one where the bottleneck occurs) that ‘‘sprays’’ buffered packets out of order when the congestion clears. The following drawing is a rework of Figure 4, showing what this hypothesized behavior would look like:

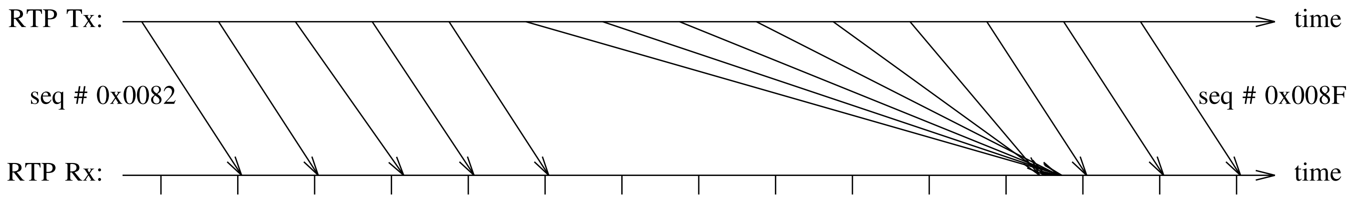


Figure 5: Hypothetical reordering of the 6 ‘‘bunched up’’ packets of Figure 4

Figure 5 above (hypothetical scenario) differs from Figure 4 (actual experience) in that the arrival times of 6 RTP packets 0x0087 through 0x008C (RTP sequence numbers from the dataset of Figure 4) have been reversed: 0x008C is hypothesized to arrive when 0x0087 actually arrived, 0x0087 is hypothesized to arrive when 0x008C actually arrived, and the 4 packets in the middle are mirrored symmetrically. Because all 6 of these packets were stuck waiting behind the same bottleneck at the same time, the fictional scenario presented in Figure 5 is at least plausible.

For the sake of completeness, let us consider a more fantastical (less likely in reality) scenario of packet reordering. Figure 6 below depicts the way beginning students of IP networking likely imagine packet reordering:

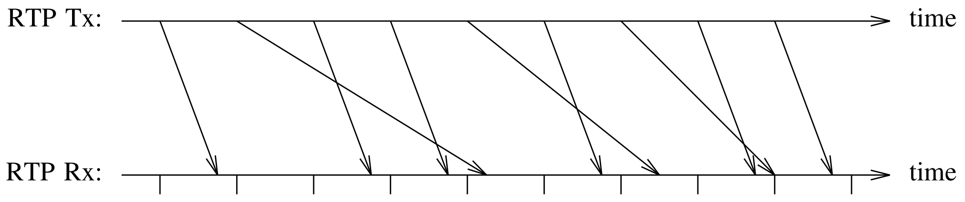


Figure 6: Unlikely form of packet reordering

In order for the fictional scenario of Figure 6 to occur, the IP network would have to behave in a way where some packets get stuck behind a delay-inducing bottleneck, yet other packets, including those that directly follow the unlucky ‘‘stuck’’ packet, are delivered without excessive delay, arriving ahead of those that got stuck. It is difficult to imagine what mechanisms could cause a real IP network to behave in such manner — but if some real IP network somewhere does indeed exhibit this theorized behavior, **twjit** should be able to handle it with appropriate tuning.

None of the examples we've examined so far include packet loss, only delay and perhaps reordering. However, each of the presented figures can be trivially modified to reflect packet loss: instead of a packet arriving late or out of order (later than a subsequently-sent packet), it never arrives at all. Readers are invited to use their imagination: take any of the arrows that represent individual RTP packets, and erase it.

2.3. Design of twjit

Having seen various scenarios of RTP flows, let us now consider what work **twjit** needs to do in order to convert a received RTP stream back to fixed timing.

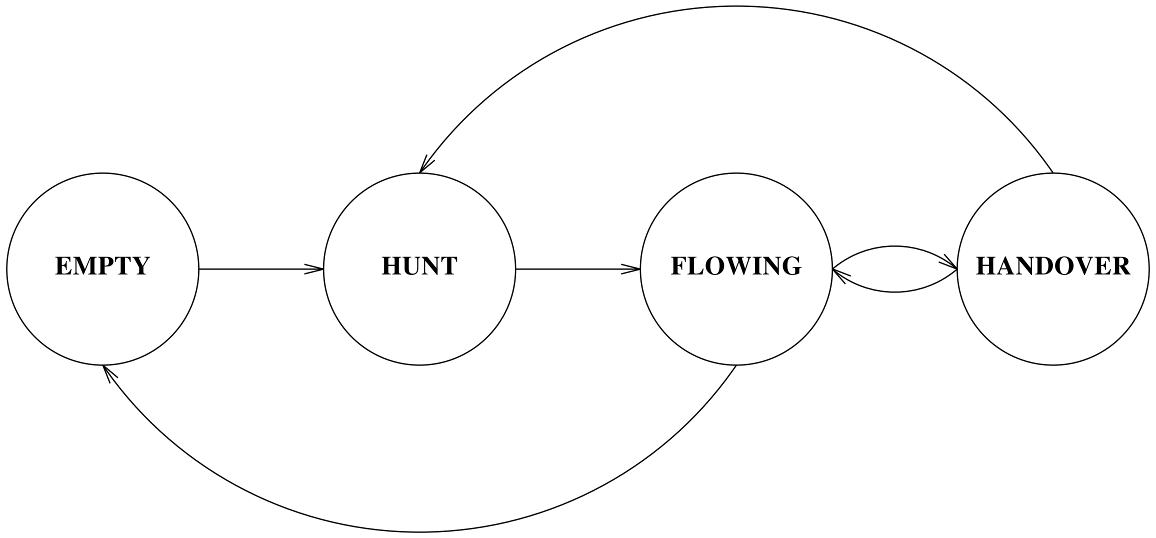
Each **twjit** instance consists of two sub-buffers (subbufs for short) and a global state variable that gives the overall state of the instance across both subbufs. Each subbuf holds a queue of received RTP packets that belong to a single RTP flow as defined in §2.1; two subbufs are needed in order to handle handovers — if the incoming RTP stream does not exhibit any handover events, a single subbuf is sufficient.

2.3.1. The major states of twjit

Each **twjit** instance as a whole, across both subbufs, is a finite state machine with 4 possible states:

EMPTY	The **twjit** instance is completely empty, neither subbuf holds any packets or any meaningful state information.
HUNT	Only one subbuf is active and valid in this state; this subbuf is non-empty — some received packets are held — but it hasn't started flowing out yet, as will be explained in the following section.
FLOWING	Only one subbuf is active and valid in this state; this subbuf is both flowing out and accepting new packets. This state is the one that holds long-term during good reception of a steady RTP flow.
HANDOVER	Both subbufs are active and valid: one is flowing out while the other receives new packets. As indicated in the name, this state is entered from the **FLOWING** state only when the received RTP stream exhibits a handover.

Possible transitions between these 4 fundamental states are as follows:



In order to understand the workings of **twjit**, let us first consider operation without handovers (SSRC never changes, and the timestamp increment from each source-emitted packet to the next always equals the samples-per-quantum constant) — in such sans-handover operation, only **EMPTY**, **HUNT** and **FLOWING** states are encountered — and then examine handover handling.

2.3.2. Structure and operation of one subbuf

Each subbuf of **twjit** holds a queue of received RTP packets that belong to a single RTP flow as defined in §2.1. In terms of memory allocation, the queue of each subbuf is implemented as a linked list of Osmocom message buffers (**msgb**s) — but this implementation detail is really only a matter of memory allocation strategy, and must **not** be misconstrued to infer what kinds of packet sequences are allowed to exist in one subbuf. In sharp contrast with a naive interpretation of what a linked list can presumably hold (any sequence of packets, without strict constraints on timestamp increment or any other aspect), the logical structure of a single **twjit** subbuf is a chain of fixed slots, where each slot corresponds to a given RTP timestamp and may be either empty or filled with a received packet.

A good physical analogy for the logical structure of a **twjit** subbuf can be found in carrier tapes that hold electronic components in tape-and-reel packaging. There is a long tape made of plastic or thick paper with regularly spaced wells, with each well intended to hold one piece of the reeled part. Whether each given well in the tape holds a component or is empty, the spacing between wells remains fixed, and a component cannot be inserted anywhere into the tape except into a designated well.

The following drawing depicts a subbuf with both filled and empty slots:

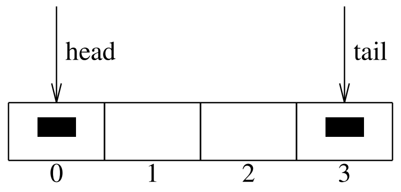


Figure 7: Basic principle of twjit subbuf

The subbuf depicted in Figure 7 has a total depth (to be defined shortly) of 4 quantum units (please recall the definition of a quantum in §1.3), the tail slot is filled as always required for a non-empty subbuf, the head slot is also filled in this example, but the other two slots are empty. (Such subbuf state may result from packet loss, and may also occur in cases of packet reordering if the packets destined for empty slots 1 and 2 may yet arrive.)

Every non-empty **twjit** subbuf has a head slot, a tail slot and a total depth. The head slot is defined by the 32-bit RTP timestamp stored in `head_ts` member of the subbuf structure: `struct twrtp_jibuf_sub` in `twrtp-proto` version or `struct twjit_subbuf` inside `twjit.c` in `twrtp-native`. If the subbuf holds a received RTP packet whose timestamp equals `head_ts`, that packet resides in the head slot; if no such packet is held, then the head slot is empty. However, packets with RTP timestamps earlier than `head_ts` **cannot** exist in a subbuf!

Every received RTP packet held in a subbuf, as well as every empty slot that can potentially be filled by a late-arriving out-of-order packet, can be viewed as existing at a certain depth. The head slot shall be regarded as depth 0, the following slot shall be regarded as depth 1, and so forth. Recall that per the fundamental design of **twjit**, each subbuf can only hold RTP packets belonging to a single flow as defined in §2.1 — thus if one quantum equals 160 timestamp units, slot 1 can only hold an RTP packet whose timestamp equals (`head_ts` + 160), slot 2 can only hold an RTP packet whose timestamp equals (`head_ts` + 320), and so forth.

Of all received RTP packets held by the subbuf, whichever packet has the newest RTP timestamp is regarded as the current tail of the subbuf, and its depth is regarded as the current tail slot. The total depth of a subbuf is defined as the depth of the tail packet plus 1; in the example of Figure 7, the tail packet has depth 3 and the total depth of the subbuf is 4. The total depth of a subbuf is also called the fill level, by analogy with fill level of a water tank.

When a new RTP packet is received, and that packet is deemed to belong to the flow already being received (same SSRC, timestamp increment meets expectations), the newly received packet is added to the current write subbuf. The insertion depth of the new packet is calculated as the newly received timestamp minus `head_ts`, divided by the number of timestamp units per quantum. If this insertion depth exceeds the current tail depth, which is the normal case, the subbuf grows (the fill level increases) and the newly added packet becomes the new tail. As a result of this operation, the tail slot of a non-empty subbuf can never be empty! Alternatively, if the insertion depth falls somewhere before the current total depth of the subbuf, the fill level stays the same and the target slot — which is expected to be empty in this case — is filled with the newly received packet. If that target slot was already filled, the new packet is discarded and an error counter is incremented, indicating duplicate Rx packets.

When an active, flowing-out subbuf is polled for output at fixed times determined by TDM or GSM Um etc, the head slot is consumed, whether it is filled or empty. If the consumed head slot was filled, that buffered packet is delivered to the fixed timing system on the output of the jitter buffer. If that slot was empty, the application on the output of the jitter buffer receives a gap in the stream. Either way, when the previous head slot is consumed, `head_ts` is incremented by the samples-per-quantum constant, the following slot becomes the new head slot, and the total depth of the subbuf decreases by 1.

There also exists a special condition in which a subbuf is empty (does not hold any buffered packets, fill level equals 0), but is still considered active. This condition can occur only in **FLOWING** state, and is covered in the respective section.

2.3.3. EMPTY and HUNT states

Upon initialization or reset, each **twjit** instance begins life in **EMPTY** state. As soon as the first valid RTP packet is received, one subbuf is initialized to hold this first packet; the total depth of this newly initialized subbuf is 1 and the slot occupied by the initial packet is both the head and the tail. The overall state of **twjit** instance transitions to **HUNT**.

The purpose of **HUNT** state is to accumulate enough received packets, necessarily belonging to a single flow as defined in §2.1, until a configured threshold is met for entry into **FLOWING** state. The critically important configuration parameter is the flow-starting fill level; it is the first number given on the `buffer-depth` vty configuration line.

The flow-starting fill level is the fill level (total depth of the sole active subbuf) required for transition from **HUNT** state into **FLOWING** state. This criterion is evaluated on every 20 ms fixed timing tick when the application polls **twjit** for a required quantum of media; as a result of this check, the **twjit** instance either delivers its first output and transitions into **FLOWING**, or returns **NULL** (‘‘sorry, I got nothing’’) to the application and remains in **HUNT** state.

The significance of this threshold parameter, and guidelines for its tuning, are best understood by looking at examples of RTP flows shown in §2.2. The minimum allowed setting for this parameter is 1; with this minimum setting, the first tick of the fixed timing system after reception of any RTP packet always causes transition into **FLOWING** state, and the packet received just prior to this transition-causing tick is delivered to the output on that tick. This setting produces the lowest possible buffer-added latency: this latency can be near-zero if the RTP packet arrived just prior to the fixed timing tick, or just under 20 ms if it arrives just after the previous tick.

However, if we look at Figures 1 and 2 in §2.2, we can see the one big problem with this lowest latency setting: the flow remains perfect under absolutely ideal conditions of Figure 1, but as soon as we enter real-world conditions as shown in Figure 2, we can easily encounter scenarios like the RTP packet with sequence number 0x0636 in that figure. If the RTP flow depicted in Figure 2 were to be received by a **twjit** instance whose flow-starting fill level is set to 1, the buffer would experience an underrun on the tick of the fixed timing system that just barely missed the slightly delayed packet; the user would then experience an equivalent of packet loss (frame erasure) at a time when no actual packet loss occurred.

For this reason, the default and generally recommended setting for the flow-starting fill level parameter is 2. With this setting, two RTP packets with properly consecutive timestamps must be received in **HUNT** state before **twjit** transitions into **FLOWING** state. The buffer-added latency will be anywhere between 20 and 40 ms, depending on the unpredictable phase alignment between arriving RTP packets and ticks of the fixed timing system on the output side of **twjit**. As long as the jitter between flight times of different packets, or its observable manifestation as interarrival jitter, remains below 20 ms (or below 16.9 ms if the receiving element is OsmoBTS whose fixed time base includes the inherent jitter of GSM Um multiframe structure), there will not be an occurence where the receiving system transforms jitter into an effective equivalent of packet loss. This amount of jitter tolerance is sufficient for most practical IP networks in this author’s experience.

But what if the IP network regularly exhibits packet delay jitter that is significantly greater than 20 ms? Suppose the network regularly exhibits conditions similar to the aberration depicted in Figure 4 — what then?

In this case the administrator of jitter-buffer-equipped network elements has to make a trade-off between two different forms of degraded user experience: either increase the flow-starting fill level setting and thereby increase the experienced latency, or live with underruns (frame erasure effectively equivalent to packet loss) whenever the IP network stops flowing smoothly and decides to ‘‘bunch up’’ packets instead. As just one example, if the amount of ‘‘bunching up’’ exhibited by the IP network were exactly as depicted in Figure 4, and the desire were to eliminate packet-loss-equivalent effects at the expense of added latency, the required flow-starting fill level setting would be 7, producing added latency between 120 and 140 ms.

2.3.3.1. Reception of additional packets in HUNT state

Every time an additional RTP packet is received when the **twjit** instance is already in **HUNT** state (after the first Rx packet that moved the state from **EMPTY** to **HUNT**), certain checks are made. The first fundamental requirement of **HUNT** state is that all queued packets belong to the same flow. If the newly received RTP packet has a different SSRC, or if it exhibits a timestamp increment that is numerically incompatible with being a member of the same flow (not an integral multiple of samples-per-quantum constant), all previously queued packets are discarded and the **HUNT** state is reinitialized anew with the just-received packet. This behavior is unavoidably necessary: the single subbuf of **HUNT** state holds packets belonging to *one* particular flow, and given the choice between a stale flow that appears to have just ended and the new flow that appears to have just begun, the new flow is clearly the correct choice.

Once the same-flow requirement is met and the newly received packet is not too old (packets whose RTP timestamps precede the current `head_ts` have to be discarded), the new packet is inserted into the sole active subbuf at its respective depth. Most of the time, this insertion will increase the total depth or fill level of this subbuf. At this point the new fill level is checked against the flow-starting fill level setting: if the flow-starting fill level has just been exceeded, packets are discarded from the head of the subbuf and `head_ts` advances forward until the remaining fill level is equal to or below the flow-starting threshold.

This trimming of the subbuf to the flow-starting fill level is necessary to ensure that the latency added by the jitter buffer will indeed be what the administrator intended to set via the tunable parameter, as opposed to potentially much higher added latency that could be caused by artifacts at flow starting time. Suppose that the IP network bunches up a significant number of packets when the sender begins transmitting them 20 ms apart, then delivers that bunch all at once, and then begins to flow evenly:

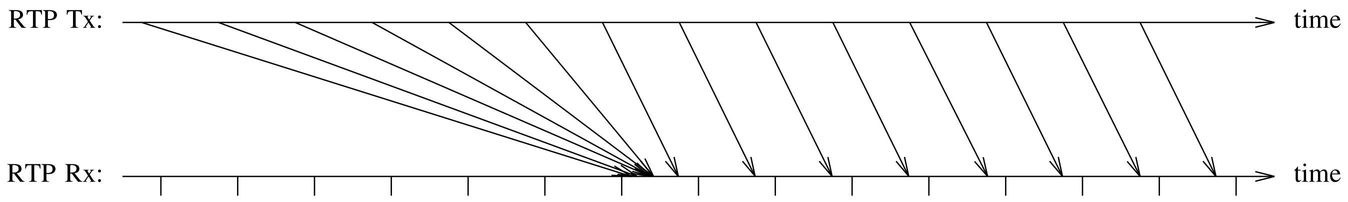


Figure 8: Packets bunched together at the beginning of flow

If the step of trimming the subbuf in **HUNT** state to the flow-starting fill level were omitted, then in the scenario depicted in Figure 8, the fill level on entry into **FLOWING** state would be 7 instead of 2 or whatever flow-starting fill level is configured by the administrator, significantly increasing the latency experienced by the user.

If the flow-starting fill level is set to 1, the total depth of the sole active subbuf in **HUNT** state will never equal anything other than 1; if the flow-starting fill level is set to 2 (the default), the total depth of the subbuf in **HUNT** state will always equal either 1 or 2, with both head and tail slots always filled. Empty slots in this sole active subbuf cannot exist when the flow-starting fill level is set to 1 or 2. However, if the flow-starting fill level is set to 3 or greater, empty slots in the subbuf in **HUNT** state become possible: both head and tail slots are still always filled in **HUNT** state, but with higher settings of the flow-starting fill level configuration parameter, it becomes possible to have empty slots in the middle, produced by packet loss or reordering.

2.3.3.2. Additional time delta guards

Let us once again consider the scenario depicted in Figure 8. The step of trimming the subbuf to the flow-starting fill level prevents induction of significantly increased latency by initial floods arriving while the **twjit** instance is still in **HUNT** state — but suppose the transition from **HUNT** into **FLOWING** occurs while an initial flood, similar to that depicted in Figure 8, is still ongoing. As covered in the following section, once the overall state of **twjit** instance is **FLOWING**, no more simple head trimming can occur: there is a much slower-acting standing queue thinning mechanism, but any extra latency that was induced at the start of the flow can only be dissipated very slowly, and with an unavoidable side effect of phase shifts in the delivered flow.

In order to produce better performance in IP network environments where scenarios like Figure 8 are expected, there is an additional check, optionally enabled per configuration, gating the transition from **HUNT** into **FLOWING** state: it is `start-min-delta` vty setting. When this optional parameter is set, it specifies the minimum required time-of-arrival delta in milliseconds between the most recently received RTP packet and the one received just prior; this minimum ToA delta must hold in order for transition from **HUNT** into **FLOWING** to be allowed.

For the sake of symmetry, there is also an optional `start-max-delta` vty setting. When this optional parameter is set, it specifies the maximum allowed ToA delta in **HUNT** and **HANDOVER** states: if the ToA delta between successively received packets exceeds this threshold, previously queued packets are discarded (regarded as remnants of a stale previous flow) and the hunt process begins anew with the latest received packet.

2.3.4. FLOWING state

When the global state of a given **twjit** instance is **FLOWING**, there is only one active subbuf, just like in **HUNT** state. Newly received RTP packets that belong to the same flow (same SSRC, timestamp increments meet expectations) are likewise added to this subbuf just as they were during **HUNT**. However, the same subbuf is also flowing out: on every tick of the fixed timing system on the output side of **twjit**, the head slot of the subbuf is consumed and `head_ts` advances accordingly.

Unlike **HUNT** state, **FLOWING** state allows the head slot of the sole active subbuf to be empty. This situation will occur if the received flow experiences a gap (packet loss, reordering or an intentional gap emitted by the RTP stream source), the last received packet before the gap is consumed, but there are still more packets at greater depth, such that the subbuf is not entirely empty. If the head slot remains empty on the fixed timing tick that consumes it, the application on the output side of **twjit** receives a gap in the stream, but this event is **not** regarded as an underrun.

2.3.4.1. Handling of underruns

It is possible for the flowing-out subbuf to be empty, but not incur an underrun just yet. Suppose the total depth (see §2.3.2) of the flowing-out subbuf equals 1 at the time of an output poll: the head slot is also the tail slot, and the previously received RTP packet consumed on this tick is the very last one received till this moment. After this output poll tick, the subbuf is empty (total depth equals 0), but it is still valid in the sense that the overall state remains **FLOWING** (does not transition to **EMPTY**) and `head_ts` is still regarded as valid, equal to the timestamp of the last delivered packet plus 160. If another RTP packet, belonging to the same flow, is received before the next output poll tick, the flow continues without underrun or any other undesirable interruptions. This situation occurs all the time in normal operation when the flow-starting fill level configuration parameter is optimally tuned, adding just enough buffering latency to handle the amount of jitter that actually occurs, but no more.

A true underrun occurs on the next output poll tick after the one that leaves the flowing-out subbuf empty while still valid. At this point the overall state of the **twjit** instance transitions to **EMPTY**. Any subsequently received RTP packet, whatever its SSRC and timestamp may be, causes a transition from **EMPTY** into **HUNT**, and the process of latching onto a flow begins anew. The application on the output of the jitter buffer will keep receiving **NULL** (‘‘sorry, I got nothing’’) starting with the output poll tick on which the underrun occurs and continuing until the new flow after the underrun (if there is one) reaches the flow-starting fill level.

For the benefit of network operations staff looking at stats counters logged upon call completion (see Chapter 3 regarding stats and analytics), the `underruns` counter is incremented not at the point where the actual underrun occurs, but upon receipt of the next RTP packet (if there is one) that makes the transition from **EMPTY** into **HUNT**. This implementation detail results in not counting the final underrun that often occurs upon call teardown, instead counting only those underrun events that are true indications of problematic network conditions or insufficient jitter buffering.

2.3.4.2. Standing queue thinning mechanism

Suppose that the beginning of an RTP flow (acquisition in **HUNT** state, then transition into **FLOWING**) happens when the IP network path experiences a spike in latency, then later that spike subsides and latency along the IP network path returns to a lower baseline. This scenario may look as follows:

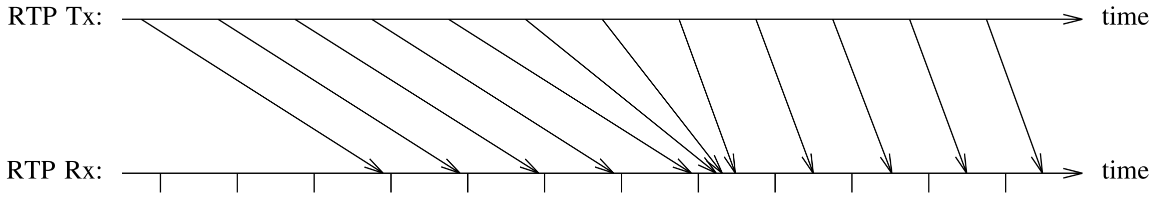


Figure 9: Period of high IP latency followed by lower latency

If the flow-starting fill level is set to 2 (the default), the total depth of the active subbuf will alternate between 1 and 2 during the initial high latency phase: increase to 2 on each RTP packet arrival, then go back down to 1 when the subsequent output poll tick consumes the packet in the head slot. However, following the transition from higher to lower IP network path latency while **twjit** is in **FLOWING** state, if the arriving packets land where they do in Figure 9, the subsequent total depth of the same active subbuf will alternate between 3 and 4: rise to 4 when ‘‘bunched up’’ packets arrive, then go down to 3 on each output poll tick and go back up to 4 as new RTP packets arrive in between those ticks.

The end result of such happenings is increased buffer-added latency — a standing queue — in the steady flow state after the latency of the IP network path went down. In the present example, the standing queue latency added as a lasting artifact of earlier network conditions at flow starting time is 40 ms — but it can be greater or smaller, in 20 ms increments, depending on how the IP network path changes between initial acquisition of a new RTP flow and its subsequent steady state.

The design of **twjit** includes a mechanism for thinning these standing queues, gradually bringing buffer-added latency down to a maximum limit set by the administrator. The controlling parameter is the high water mark fill level; it is the second number given on the `buffer-depth` vty configuration line. This parameter must be greater than or equal to the flow-starting fill level, but it takes effect only in **FLOWING** state, not in **HUNT**. Any time the total depth of the flowing-out subbuf exceeds this high water mark on an output poll tick, tested just before consuming the head slot and any RTP packet contained therein, the standing queue thinning mechanism kicks in. This mechanism deletes every *N*th packet from the stream being thinned, where *N* is the `thinning-interval` parameter set in vty config, for as long as the total depth of the flowing-out subbuf exceeds the high water mark fill level. More precisely, every *N*th output poll tick that happens in the state of high water mark being exceeded advances the head slot by two quantum units instead of one, with the first consumed head slot always discarded and the second consumed head slot passed to the output. This operation remains the same irrespective of whether each of the two thus consumed head slots holds a previously received RTP packet or is empty.

The act of deleting a quantum from the middle of an ongoing stream of speech or CSData is always disruptive: in the case of speech, a 20 ms quantum, potentially in the middle of a speaker talking, suddenly disappears; in the case of CSData, the effect may be even worse with a potentially important data chunk likewise disappearing, plus a 20 ms phase shift in the stream is always incurred. However, such disruptions are the only way to bring down a standing queue, whose added latency is another form of evil — thus as usual, engineering is all about trade-offs and compromises.

The default configuration for **twjit** is `buffer-depth 2 4` and `thinning-interval 17`: the default high water mark fill level is 4, and the default thinning interval is to delete every 17th packet, i.e., delete one 20 ms quantum every 340 ms. The exact scenario depicted in Figure 9 will not invoke the standing queue thinning mechanism with these default settings: in this depicted scenario, the total depth of the active subbuf rises to 4 at its highest, which is also the default high water mark fill level. However, if this high water mark setting is lowered or if the latency spike at the beginning of the flow is more substantial, then the standing queue thinning mechanism will kick in when the IP latency spike subsides.

The default thinning interval of deleting every 17th packet was chosen based on these considerations:

a) 340 ms is long enough to where 20 ms quantum deletions spaced this far apart should be tolerable, yet short enough to where a standing queue would be reduced to the high water mark in reasonable time;

b) 17 is a prime number, thereby reducing the probability that the thinning mechanism will interfere badly with intrinsic features of the stream being thinned.

The default high water mark fill level was chosen so as to provide some margin above the flow-starting fill level (allow IP network path latency variations without needless thinning followed by underrun and reacquisition), while still maintaining a constraint against unbounded growth of a standing queue. As always, the optimal engineering trade-off will depend very strongly on the actual characteristics of the IP network environment on top of which a GSM network or IP-PSTN system is being built, hence careful attention is required from the managing operator.

2.3.4.3. Guard against time traveler packets

Every time a new RTP packet is received in **FLOWING** state, a series of checks are made to answer this question: should the newly received packet be treated as a continuation of the current flow, or should it be treated as belonging to a new flow and thus a handover event per §2.1? The first check is SSRC comparison: if the newly received RTP packet has a different SSRC than the currently active flow, it is a handover.

The RTP timestamp is checked next. In order to handle arbitrary starting 32-bit timestamps and wraparound of the absolute 32-bit timestamp value at any point, **twjit** code computes the difference between current subbuf `head_ts` (subtrahend) and the newly received timestamp (minuend), and treats this difference as a signed 32-bit integer. If this difference is negative, the newly received packet is treated as a stale (too old) one, received with so much delay that it can no longer be accepted: `too_old` stats counter is incremented, and the packet in question is discarded without further processing. After this check the timestamp increment, now confirmed to be non-negative, is checked to see if it is an integral multiple of the samples-per-quantum constant, which is usually 160. If this integral multiple constraint is violated, the new packet cannot belong to the same flow as the currently active one, and it is necessary to invoke handover handling.

After these two checks, one final check is needed for robustness: a guard against time traveler packets. If the increment between current `head_ts` and the timestamp field in the newly received RTP packet is positive after wraparound handling, if it is an integral multiple of the samples-per-quantum constant, but it is excessively large (at 8000 timestamp units per second, the largest possible post-wraparound-handling RTP timestamp increment is just over 3 days into the future), such aberrant RTP packets are jocularly referred to as time travelers.

Assuming that actual time travel either does not exist at all or at least does not happen in the present context, we reason that when such ‘‘time traveler’’ RTP packets do arrive, we must be dealing with the effect of a software bug or misdesign or misconfiguration in whatever foreign network element is sending us RTP. In any case, irrespective of the cause, we must be prepared for the possibility of seeming ‘‘time travel’’ in the incoming RTP stream. We implement an arbitrary threshold: if the received RTP timestamp is too far into the future, we treat that packet as the beginning of a new flow, same as SSRC change or non-quantum timestamp increment, and invoke handover handling.

The threshold that guards against time traveler packets has 1 s granularity, which is sufficient for its intended purpose of catching gross errors. It is set with `max-future-sec` vty configuration line. The default value is 10 s: very generous, perhaps overly so, to networks with really bad latency.

2.3.5. Handling of packet loss and gaps

The design of RTP makes it impossible to distinguish between packet loss and intentional gaps in real time: if a packet fails to arrive at the time when it is expected and needed on the receiving end, the receiver has no way of knowing *at that moment* whether the cause of this lack of packet arrival is an intentional gap emitted by the sender or packet loss along the way. This distinction can be made later, after the fact: when subsequent packets arrive, the receiver can examine the sequence number field in the RTP header and thereby determine which of the two events (intentional gap or packet loss) happened previously. However, because this knowledge is not available at the time when it would be needed, **twjit** makes no distinction between these two possibilities outside of analytics. For the purpose of mapping received RTP packets to ticks of the fixed timing system on the output side of **twjit**, any intentional gaps in the incoming RTP stream are treated indistinguishably from packet loss. (Please recall from §1.2.1 that **twjit** is designed for use with continuous streaming, not intentional gaps.) The operational (as opposed to analytics) part of **twjit** looks only at SSRC and timestamp fields in the RTP header; it does not consider the sequence number at all.

The actual effect of a gap in the received RTP stream, whether intentional or caused by packet loss, depends on the size of the gap (how many consecutive packets are lost or omitted) and jitter buffer conditions at the time of its occurrence. The critical question is whether or not the gap in RTP reception incurs an underrun:

- If the gap is small enough, or the running depth of the jitter buffer is high enough, to where no underrun occurs, then the gap is presented to the application on the output side of **twjit** without distortion: the application will receive **NULL** (absence of received packet) in those quanta whose corresponding RTP packets (corresponding per RTP timestamp) were not received, while all packets which did arrive will be delivered correctly, each at its respective quantum point. No phase shift is incurred in the received stream of media.
- If the gap results in an underrun, subsequent received packets after this gap will proceed as acquisition of a new flow, passing through **HUNT** state before entering **FLOWING**. Preservation of phase cannot be guaranteed under such conditions, i.e., the gap as perceived by the fixed timing application may be lengthened or shortened compared to the actual number of RTP packets lost or omitted.
- If a gap incurs an underrun, then there is a single RTP packet following this gap, then another gap, the lone RTP packet between the two gaps will be dropped, assuming the default configuration with flow-starting fill level set to 2. This drop occurs because a single RTP packet following an underrun is not sufficient to establish a new flow when the flow-starting fill level is greater than 1, and when another RTP packet arrives after the second gap, the first between-gaps packet will be too stale. This failure scenario is the reason why the combination of **twjit**, DTX and intentional gaps will not work.

If **twjit** is deployed in an IP network environment where packet loss occurs frequently enough to be a concern, it may be necessary to increase the amount of buffering (by increasing the flow-starting fill level) so that packet loss events do not turn into underruns. However, intentional gaps are always bad in principle and should be avoided — enable continuous streaming instead.

2.3.6. Handling of packet reordering

As already covered in §2.2, we (Themyscira Wireless) have no operational experience with packet reordering, as it is a behavior which is not exhibited by IP networks in our part of the world. Let us consider nonetheless how **twjit** would handle theoretically envisioned cases of packet reordering that were presented in that section.

The hypothetical scenario depicted in Figure 5 calls for exactly the same **twjit** configuration as the real world scenario of Figure 4. If the IP network frequently exhibits effects like those depicted in Figure 4 and Figure 5, the flow-starting fill level would need to be set to 7. As long as episodes of packets bunched together, with or without reordering, are separated by periods of smooth packet flow, **twjit** would proceed through its acquisition stage (**HUNT** state) in one of these smooth flow periods, and then episodes of the form depicted in Figure 4 or Figure 5 would be handled gracefully, without any loss or distortion of transported media.

The more fantastical scenario of Figure 6 would be handled well by setting the flow-starting fill level to 4. Visualizing the state of the sole active subbuf after each RTP packet arrival and after each output poll tick in that figure is left as an exercise for the reader — however, each media quantum carried by each of the packets shown in the figure will be delivered to the application on the output side of **twjit** in the correct order, without any loss or distortion.

2.3.7. Handling of handovers

As covered in §2.1, a handover in **twjit** terminology is a transition from one RTP flow to the next, within the context of a single RTP stream. A handover occurs when the incoming RTP stream exhibits a change of SSRC, a timestamp increment that is not an integral multiple of the samples-per-quantum constant, or a ‘‘time travel’’ event as described in §2.3.4.3.

In order to be treated as a handover by **twjit**, the newly received RTP packet that breaks the previous flow in one of the just-listed ways must arrive while the previous flow (the one it breaks from) is still active, while the state of the **twjit** instance is **FLOWING**. If a handover happens after the previous flow underruns, such that the **twjit** instance is in **EMPTY** state when the first packet of the new flow arrives, acquisition of the new flow proceeds via **HUNT** state in the same way whether this new flow is continuous or discontinuous with the previous one. Similarly, if a flow discontinuity (the kind that would be treated as a handover if it occurred in **FLOWING** state) occurs in **HUNT** state, it is handled by reinitializing the **HUNT** state, without entering the special **HANDOVER** state, as detailed in §2.3.3.1.

True handover handling happens when a flow-breaking RTP packet arrives in **FLOWING** state. This event causes a transition into the dedicated **HANDOVER** state, described here. In this state both subbufs of **twjit** are active and valid: the subbuf that was active in **FLOWING** state continues to flow out, while the other subbuf is initialized for the new flow just like in **HUNT** state. Accounting for **HANDOVER** state, each **twjit** instance has a potentially valid write subbuf and a potentially valid read subbuf, breaking down as follows:

- In **EMPTY** state, there is neither a valid write subbuf nor a valid read subbuf;
- In **HUNT** state, there is a valid write subbuf, but no valid read subbuf;
- In **FLOWING** state, the sole active subbuf is both the write subbuf and the read subbuf;
- In **HANDOVER** state, the read subbuf and the write subbuf are different, and each is valid.

Once **HANDOVER** state has been entered, the code path that handles incoming RTP packets operates like it does in **HUNT** state, while the output path that executes on ticks of the fixed timing system operates like it does in **FLOWING**, each operating on its respective subbuf. There are two possible exit conditions from this state:

1) If the new write subbuf reaches ready state (the same criterion as applied for transition from **HUNT** to **FLOWING**, covered in §2.3.3) before the old read subbuf underruns, the **twjit** instance transitions from **HANDOVER** back into **FLOWING** state. Any packets that remain in the old read subbuf are discarded, and the new write subbuf becomes the sole active subbuf for both reading and writing.
2) If the old read subbuf underruns before the new write subbuf is ready to start flowing out, a handover underrun occurs (same as a regular underrun, but increments a different stats counter) and **twjit** state transitions to **HUNT**. This handover underrun will occur if the new write subbuf does not become ready quickly enough, as the old read subbuf no longer receives any new packets in **HANDOVER** state.

2.3.8. Additional notes

Some additional notes about **twjit** design that don't fit anywhere else:

- Neither the payload type nor any of payload content are checked by **twjit**: all payload handling is the responsibility of the application.
- RTP packets with zero-length payloads are treated as no different from other valid packets; such packets may be needed to ensure continuous streaming, as covered in §1.2.1.
- Only SSRC and timestamp fields in the RTP header are considered for the purpose of mapping received RTP packets to ticks of the fixed timing system on the output of **twjit**. The sequence number field is examined only for analytics (see Chapter 3), but not for actual operation.
- The marker (**M**) bit in the RTP header is ignored by **twjit**; it does not factor into any jitter buffer algorithm decisions about flow transitions, handovers, readiness to start flow or thinning of standing queues. This design decision is expected to be controversial, but the author of **twrtp** and **twjit** stands by this decision. The **M** bit appears to have been included in RTP with the intent of serving a purpose in a paradigmatic universe built around the practice of intentional gaps — it marks the first packet of a ''talkspurt'' after an intentional gap — but **twjit** is designed for a different paradigmatic universe. In our (Themyscira Wireless) paradigmatic universe, RTP transport is used instead of physical T1/E1 circuits only due to economic pressures (Internet is essentially cost-free while physical circuits are expensive), **not** out of any philosophical love for packet-based transmission, and we favor continuous streaming as described in §1.2.1. In the environment of continuous streaming, RTP's **M** bit serves no useful purpose.

2.4. Summary of configuration parameters

Applications that use **twjit**, usually as part of **twrtp**, are expected to also use Osmocom vty system for configuration. All tunable configuration parameters for **twjit** are gathered into a config structure, named `struct twrtp_jibuf_config` in `twrtp-proto` version or `struct twna_twjit_config` in `twrtp-native`; this config structure must be provided every time a **twjit** instance is created. Every application that uses **twrtp** with **twjit** is expected to maintain one or more of these config structures, accessible to tuning via vty. Multiple **twjit** configuration parameter sets in one application may be needed if the application creates different kinds of RTP endpoints that may need different **twjit** tunings: for example, `tw-border-mgw` has one **twjit** configuration parameter set for GSM RAN side and another for IP-PSTN side. Vty configuration for **twjit** looks like this (excerpt from `tw-border-mgw.cfg`):

```
twjit-gsm
 buffer-depth 2 4
 thinning-interval 17
 max-future-sec 10
twjit-pstn
 buffer-depth 2 4
 thinning-interval 17
 max-future-sec 10
```

All numbers in the example above are defaults for the respective settings. Individual settings are as follows:

- `buffer-depth` line controls the flow-starting fill level (first number) and the high water mark fill level (second number), parameters that affect the amount of latency added by **twjit** in return for tolerance to jitter and longer-term variations in IP network path delay. The flow-starting fill level is described in detail in §2.3.3; the high water mark fill level is described in §2.3.4.2.
- `thinning-interval` setting controls the interval at which quantum units are deleted from the received stream when the standing queue thinning mechanism kicks in — see §2.3.4.2 for the detailed description.
- `max-future-sec` setting adjusts the guard against time traveler packets, described in detail in §2.3.4.3.
- `start-min-delta` and `start-max-delta` optional settings (each can be set or unset) allow the managing operator to set additional timing constraints that need to be met in order to start a new flow: see §2.3.3.2 for full details.

3. Stats and analytics

Every **twjit** instance maintains a set of statistical counters, collected into `struct twrtp_jibuf_stats` in `twrtp-proto` version or `struct twna_twjit_stats` in `twrtp-native`. The purpose of these counters is to assist network operations staff: applications that use **twrtp** with **twjit** are expected to provide vty introspection commands that display these statistical counters in real time for ongoing calls or connections, and then log any non-zero counters at call completion. Implementors of new applications are encouraged to examine the source for `tw-border-mgw`, C modules named `end_stats.c` and `introspect.c`, for an example of how these functions should be implemented.

The rest of this chapter provides a description of every counter in **twjit** stats structure.

3.1. Normal operation counters

The following counters record events that are expected to occur in normal operation, in the absence of any errors or adverse conditions:

`rx_packets`	This counter increments for every packet that was fed to **twjit** input and passed the basic RTP header validity check.
`delivered_pkt`	This counter increments for every packet that was pulled from the head of a read subbuf for delivery to the fixed timing application on the output side of **twjit**.
`handovers_in`	This counter increments when **twjit** state transitions from **FLOWING** into **HANDOVER**, as described in §2.3.7.
`handovers_out`	This counter increments when **twjit** state transitions from **HANDOVER** back to **FLOWING** *without* incurring a handover underrun first, i.e., when the new (post-handover) packet flow becomes ready to flow out before the old one underruns.

3.2. Adverse event counters

The following counters record events that are undesirable, but not totally unexpected:

`too_old`	This counter increments when an RTP packet received in **FLOWING** state has a timestamp that precedes the active subbuf's `head_ts`. This event can only occur if some packet reordering took place, such that an earlier-sent packet arrived later than a later-sent one, or if the buffer is in ''pre-underrun'' state (see §2.3.4.1) and the very last RTP packet that just flowed out is duplicated.
`underruns`	This counter increments when a **FLOWING** state underrun occurs, followed by reception of at least one post-underrun RTP packet that is then treated as the beginning of a new flow — see §2.3.4.1.
`ho_underruns`	This counter increments when an underrun occurs in **HANDOVER** state, i.e., when the previous flow underruns before the new one is ready to start flowing out.
`output_gaps`	This counter increments for every gap in the output stream from **twjit** that occurs in **FLOWING** or **HANDOVER** state *without* an underrun, i.e., with the flow still continuing and further queued packets present past the gap.
`thinning_drops`	This counter increments when the standing queue thinning mechanism described in §2.3.4.2 deletes a quantum from the stream delivered to the application on the output of **twjit**. Please note that `delivered_pkt` or `output_gaps` is still incremented for the quantum that is pulled from the read subbuf, but then artificially deleted by the thinning mechanism.

3.3. Error counters

The following counters record truly unusual and unexpected error events:

`bad_packets`	This counter increments when a packet that was fed to **twjit** input is too short for RTP (shorter than the minimum RTP header length of 12 bytes) or has an unknown value in the RTP version field.
`duplicate_ts`	This counter increments when **twjit** attempts to add a newly received RTP packet to the active-for-write subbuf, but a previous packet is already held with the same timestamp and thus at the same depth position. For the sake of implementation simplicity in this error case that should not occur in a correctly working system, **twjit** drops the new packet and keeps the old one.

3.4. Independent analytics

In addition to its main function of mapping received RTP packets to ticks of the fixed timing system on its output, **twjit** performs some ''raw'' analytics on the stream of RTP packet it receives. These analytic steps are independent of **twjit** algorithm details, of any configuration settings summarized in §2.4, and independent of what happens on the output (fixed timing) side of **twjit** — thus they bear no direct relation to **twjit** state transitions, subbuf conditions and so forth. Instead these analytics depend only on the shape of the incoming RTP stream itself, same as if an analyst were looking at a pcap file after the fact. Some of these analytic steps are performed in order to gather information for the purpose of generating RTCP reception report blocks (see Chapter 5), but some simple analytic steps are done solely to produce some additional stats counters that are expected to be valuable to network operations staff. The following counters are maintained as part of these independent analytics:

`ssrc_changes`	This counter increments when the received RTP stream exhibits a change of SSRC, or more precisely, every time an RTP packet arrives whose SSRC differs from that of the packet received just prior.

All following counters record events that occur within a same-SSRC substream:

`seq_skips`	This counter increments every time a packet is received whose sequence number increment (over the packet received just prior) is positive and greater than 1. Such

	occurrence indicates either packet loss or reordering in the IP network.
`seq_backwards`	This counter increments every time a packet is received whose sequence number goes backward, relative to the packet received just prior. Such occurrence indicates packet reordering in the IP network.
`seq_repeats`	This counter increments every time a packet is received whose sequence number is the same as the packet received just prior. Such occurrence indicates packet duplication somewhere.
`intentional_gaps`	This counter increments every time a packet is received whose sequence number increments by 1 over the packet received just prior, indicating no packet loss or reordering at the hands of the transited IP network, the timestamp increment is positive and an integral multiple of the samples-per-quantum constant, but this increment does not equal exactly one quantum. Such occurrence indicates that the RTP stream sender emitted an intentional gap.
`ts_resets`	This counter increments every time a packet is received whose sequence number increments by 1 over the packet received just prior, but the timestamp relation between this packet and the previous one is neither the expected single quantum increment nor an increment of multiple quanta consistent with an intentional gap.
`jitter_max`	This reporting variable is not a counter, but a quantitative measure. It reports the highest interarrival jitter that was encountered within the present same-SSRC substream, measured as prescribed by RFC 3550: the absolute value of the difference between the timestamp delta of two adjacently-received packets and the Δ in time of arrival, converted from seconds and nanoseconds to RTP timestamp units.

4. RTP endpoint functionality

The previous two chapters covered **twjit**, the jitter buffer component of ThemWi RTP endpoint implementation. However, this **twjit** layer is not expected to be used directly by applications: an application that needs to implement an RTP endpoint will need an endpoint implementation that actually sends and receives RTP packets, and possibly RTCP as well. The top layer of ThemWi RTP endpoint library, named **twrtp**, provides this functionality.

The API to this **twrtp** layer has been cleaned up significantly between initial `twrtp-proto` and the polished version in `twrtp-native`. (The version submitted as a merge candidate to Osmocom is similar to `twrtp-native` in this regard.) For ease of exposition, this chapter will describe `twrtp-native` version of the API. Programmers who need to use `twrtp-proto` should be able to figure it out by reading the source.

4.1. Endpoint life cycle

Every **twrtp** endpoint is represented by opaque `struct twna_twrtp`, which is a talloc context. These endpoints are created with `twna_twrtp_create()` and freed with `twna_twrtp_destroy()`. Every **twrtp** instance (`struct twna_twrtp`) owns the two UDP sockets that are bound to RTP and RTCP ports, their corresponding `struct osmo_io_fd` instances, the subordinate **twjit** instance if one exists, and any buffered packets. All of these resources are released upon `twna_twrtp_destroy()`.

4.2. Supplying UDP sockets for RTP and RTCP

For every **twrtp** endpoint, there is one file descriptor referring to the UDP socket to be used for RTP, and another file descriptor referring to the UDP socket to be used for RTCP. How do these UDP sockets and file descriptors come into being? Two ways are supported:

1) In self-contained Osmocom applications where **twrtp** is to be made available as an alternative to Belledonne **ortp**, as well as `tw-e1abis-mgw` fitting in the place of OsmoMGW-E1, `twna_twrtp_bind_local()` or `osmo_twrtp_bind_local()` creates both sockets and binds them to a specified IP:port address, supporting both IPv4 and IPv6 and automatically incrementing the port number by 1 for RTCP.

2) In Themyscira Wireless CN environment, there is a separate daemon process that manages the pool of local UDP ports for RTP+RTCP pairs, and that daemon passes allocated sockets to its clients via UNIX domain socket file descriptor passing mechanism. A network element that uses this mechanism will receive a pair of file descriptors for already-bound UDP sockets from `themwi-rtp-mgr`; these two already-allocated and already-bound UDP socket file descriptors are then passed to `twna_twrtp_supply_fds()`.

Either way, the two UDP sockets and their file descriptors are then owned by the containing **twrtp** instance, and will be closed upon `twna_twrtp_destroy()`.

4.3. RTP remote address

The two UDP sockets for RTP and RTCP always remain unconnected at the kernel level — instead the notion of the remote peer address is maintained by **twrtp** library. This remote address needs to be set with `twna_twrtp_set_remote()`; until it is set, no RTP or RTCP packets can be sent or received. Once the remote address is set, the library will send outgoing RTP and RTCP packets to the correct destination, and the same remote address is also used to filter incoming packets: incoming RTP and RTCP packets are accepted only if the UDP source address matches the currently set remote peer. This remote peer address can be changed as needed throughout the lifetime of the RTP endpoint.

4.4. RTP receive path

Applications using **twrtp** can receive incoming RTP packets in two ways: with or without **twjit**. Every application that uses **twrtp** must decide, at the time of endpoint creation via `twna_twrtp_create()`, whether or not this endpoint should be equipped with **twjit**; if a **twjit** instance is needed along with **twrtp**, the application must provide a `struct twna_twjit_config` — see §2.4.

API functions for receiving incoming RTP traffic via **twjit**, namely `twna_twrtp_twjit_rx_ctrl()` and `twna_twrtp_twjit_rx_poll()`, can be used only on **twrtp** endpoints that were created with **twjit** included — however, the other RTP Rx API, namely `twna_twrtp_set_raw_rx_cb()` for non-delayed unbuffered Rx path, is available with all **twrtp** endpoints. Applications are allowed to mix **twjit** and raw Rx paths: if a raw Rx callback is set, that callback function is called first for every received packet, and it can either consume the **msgb** passed to it, or leave it alone. If the callback function returns **true**, indicating that it consumed the **msgb**, **twrtp** Rx processing ends there; if it returns **false**, or if there is no raw Rx callback installed, then the packet is passed to **twjit** if present and enabled, otherwise it is discarded.

The ability to use both **twjit** and the non-delayed unbuffered Rx path at the same time is particularly useful for speech transcoder implementations that support AMR codec on the RAN side: such TC will use **twjit** to feed the incoming RTP stream to the speech decoder function that runs on fixed timing, but the non-delayed Rx path can also be used to ‘‘peek’’ at received RTP packets as they come in and extract the CMR field — to be fed to the speech encoder element, which is separate from the speech decoder fed via **twjit**.

4.5. RTP Tx output

The primary purpose of **twrtp** library is to facilitate implementation of bidirectional interfaces between an RTP stream and a fixed timing system such as GSM Um TCH or T1/E1 TDM. Most of the work is in receiving the incoming RTP stream and mapping incoming RTP packets to ticks of the fixed timing system, as covered in Chapter 2 — however, output from the fixed timing system to RTP also requires some consideration.

4.5.1. Choice of output SSRC

A random Tx SSRC is assigned to each **twrtp** endpoint when it is created with `twna_twrtp_create()`. No loop detection or SSRC collision logic is implemented: if it so happens that both ends of the RTP link pick the same SSRC, no adverse effects will occur for **twrtp**. If the foreign RTP implementation on the other end does object to SSRC collisions and applies some logic along the lines of RFC 3550 §8.2, it is welcome to change its SSRC: on **twrtp** receiving end such incoming SSRC changes will be treated by **twjit** like any other handover. However, the SSRC emitted by the local **twrtp** end will remain the same throughout the lifetime of the endpoint.

4.5.2. Starting, stopping and restarting Tx flow

The initial Tx flow is established when the application calls `twna_twrtp_tx_quantum()` for the first time on a given endpoint. At this point the initial RTP timestamp for this Tx flow is set, based on the current UTC time (CLOCK_REALTIME) reading plus an optional random addend. The current UTC reading at the moment of Tx flow start is used, rather than a purely random number, for consistency with timestamp computation in the case of restart, as we shall see momentarily. Once the flow is started in this manner, the application must commit to calling `twna_twrtp_tx_quantum()` or `twna_twrtp_tx_skip()` every 20 ms without fail; each of those calls will increment the timestamp by the samples-per-quantum constant, usually 160.

If this Tx flow continues uninterrupted for the lifetime of the RTP endpoint, the receiving end will see a timestamp increment of one quantum in every successive packet, forming a perfectly continuous flow. In this case the starting absolute value of the RTP timestamp does not matter at all; the UTC-based starting timestamp derivation used by **twrtp** is indistinguishable from a random number. But what if the sending endpoint needs to interrupt and then restart its output?

A dedicated mechanism is provided for such restarts after interruption. If an application stops emitting packets via `twna_twrtp_tx_quantum()` but later restarts, it must call `twna_twrtp_tx_restart()` any time between the last quantum Tx call of the old flow and the first such call of the new flow. When `twna_twrtp_tx_quantum()` is called with the internal restart flag set, a timestamp reset is performed. The new timestamp is computed from the current UTC reading just like on initial Tx start, but then the resulting delta relative to timestamps of the previous flow is checked, and the new timestamp may be adjusted so that the timestamp increment seen by the remote peer is always positive per 32-bit timestamp wraparound rules, and is **not** an integral multiple of the samples-per-quantum constant. The resulting effect is that the far end will see a discontinuity which **twjit** would treat as a handover, yet the increment of the RTP timestamp over this discontinuity gap is a best effort approximation of the actual time difference.

4.5.3. Ability to emit intentional gaps

As already covered in other parts of this document, Themyscira Wireless philosophy is opposed to the practice of intentional gaps in an RTP stream, and **twjit** receiver performs suboptimally in the presence of such. However, **twrtp** must be able to function as a drop-in replacement for Belledonne **ortp** library in the context of OsmoBTS application; OsmoBTS defaults to intentional gaps unless `rtp continuous-streaming` vty option is set. Therefore, **twrtp** library provides the necessary support for emitting intentional gaps: it is `twna_twrtp_tx_skip()` function.

4.5.4. Setting RTP marker bit

In §2.3.8 we have covered that our **twjit** receiver ignores the **M** bit in the RTP header. However, this bit still needs to be set one way or the other in RTP packets which we send out, hence **twrtp** provides the necessary mechanism.

In an environment that uses continuous streaming (no intentional gaps), Themyscira recommendation is to set **M** bit to 1 on the very first emitted RTP packet and on the first packet following a restart (induced discontinuity), and set it to 0 on all other packets. To produce this behavior with **twrtp**, pass the first two Boolean arguments to `twna_twrtp_tx_quantum()` as **false** and **true**. For other policies with respect to setting the **M** bit (for example, as would be needed when using **twrtp** in the place of **ortp** in OsmoBTS), see `twna_twrtp_tx_quantum()` API documentation.

4.6. No-delay forwarding between RTP endpoints

The present library supports building applications that forward RTP packets from one **twrtp** endpoint to another without passing through **twjit** and thus without adding buffering delay. To establish such a shortcut path, register a raw (unbuffered) RTP receiver on one endpoint via `twna_twrtp_set_raw_rx_cb()`, and in that callback function, pass the **msgb** to `twna_twrtp_tx_forward()`. Such cross-connect may be applied in one or both directions as needed.

Each endpoint that is involved in such cross-connection can switch at any time between forwarding packets as just described and emitting internally generated in-band tones or announcements; the latter should be emitted with `twna_twrtp_tx_quantum()`, and be sure to also call `twna_twrtp_tx_restart()` between separate episodes of locally generated output. The receiving RTP end will see handover events as SSRC switches between the one emitted by **twrtp** and the one coming from the other remote party. Actual timing will also switch, as there is no realistic way that your own 20 ms timing for announcement playout will exactly match the timing of the RTP stream switched from the other remote party.

5. Support for RTCP

ThemWi RTP endpoint library includes a built-in receiver and parser for RTCP packets: it knows how to parse SR and RR packets, it extracts information from RTCP reception report blocks that may be useful to the application, and it saves information from the sender info portion of SR packets for use in generating its own reception report blocks. The library also includes a facility for generating its own RTCP packets, either SR or RR, using information from **twjit** to fill out the reception report block. This chapter describes all library facilities related to RTCP, across both **twrtp** and **twjit** layers.

5.1. Collection of RR info in twjit

The analytic function of **twjit**, described in §3.4, collects not only the set of statistical counters described in that section, but also a set of info for the purpose of generating RTCP reception reports. In `twrtp-native` version of **twrtp** and **twjit**, these tidbits are collected into `struct twna_twjit_rr_info`; this structure is to be retrieved from a **twjit** instance via `twna_twjit_get_rr_info()`. RTCP sender function in the upper layer of **twrtp** uses this RR info structure to fill out the reception report block.

5.2. RTCP receiver in twrtp

Every packet that arrives at a **twrtp** endpoint's RTCP port, coming from the correct source address that matches the current remote peer, is parsed by the library. The parser captures SR and RR information; since these two groups of data are captured for different purposes, they are best studied separately.

5.2.1. Extraction of SR info from received RTCP packets

If the received RTCP packet is a correctly formed SR packet per RFC 3550 §6.4.1, **twrtp** notes that an SR packet was received, captures the local time (CLOCK_MONOTONIC) of its arrival, notes the SSRC of the SR sender, and saves the middle 32 bits of the 64-bit NTP timestamp in the SR. This saved information will be used later if and when this **twrtp** instance generates its own reception report.

5.2.2. Extraction of RR info from received RTCP packets

If the received RTCP packet is either SR or RR (either packet type is allowed to carry anywhere from 0 to 31 reception report blocks), the RTCP receiver in the library checks every included RR block to see if it describes our Tx SSRC, i.e., the one that was assigned as described in §4.5.1. If such SSRC-matching RR block is seen, **twrtp** sets a flag noting so, and captures two words of useful info from the report: the word describing packet loss and the word that expresses interarrival jitter. Both words are described in RFC 3550 §6.4.1. These words are captured for retrieval by the application, to be made accessible for vty introspection and logged upon call completion, along with locally collected stats as described in Chapter 3.

5.3. Emitting RTCP packets

The library is capable of generating 3 forms of RTCP packet:

- SR packet containing a single RR block;
- SR packet containing no RR block;
- RR packet containing a single reception report block.

The following sections describe how these RTCP packets may be generated and emitted.

5.3.1. Setting SDES strings

RFC 3550 §6.1 stipulates that every RTCP SR or RR packet also needs to include an SDES block, containing at least a CNAME string. These SDES strings (the mandatory CNAME and any optional ones) are set with `twna_twrtp_set_sdes()` API function; the application must call this function before any SR or RR packets can be emitted.

5.3.2. Emitting SR packets

In this library implementation, SR packets can be emitted in only one path: together with locally generated (not forwarded) RTP data output, as a result of the application calling `twna_twrtp_tx_quantum()`. There are two ways to cause `twna_twrtp_tx_quantum()` to emit RTCP SR in addition to its regular RTP data packet carrying its normally emitted quantum of media:

- The application can call `twna_twrtp_set_auto_rtcp_interval()` and thus configure the library to automatically emit an RTCP SR packet after every so many regular RTP data packets sent via `twna_twrtp_tx_quantum()`.
- The application can control directly which calls to `twna_twrtp_tx_quantum()` should emit RTCP SR via the last Boolean argument to this function.

Whichever condition is used to trigger emission of RTCP SR packet, the decision as to whether or not this SR packet will include an RR block in addition to the required sender info is made by the library. This RR block will be included if and only if:

a) this **twrtp** instance is equipped with **twjit**, and

b) at least one valid RTP packet has been received by this **twjit** instance, producing the necessary SSRC-keyed RR info structure.

The first 3 words in the RR block (the packet loss word, extended highest sequence number received and interarrival jitter) are always filled based on the info provided by **twjit** via `struct twna_twjit_rr_info`. However, the last 2 words (LSR and DLSR) are filled based on info captured by **twrtp** layer's RTCP receiver, as described in §5.2.1. If an SR was previously received by this **twrtp** endpoint and the sender of that SR had the same SSRC as the one for which we are producing our reception report (the SSRC in `struct twna_twjit_rr_info`), then information from that received SR (its time of arrival and saved NTP timestamp bits) is used to fill LSR and DLSR words in the generated RR block. Otherwise, these two words are set to 0.

5.3.3. Emitting standalone RR packets

In most RTCP-enabled RTP applications, it is most useful to emit SR packets and convey reception report blocks as part of them. However, **twrtp** library also provides a way to emit standalone RR packets, which can be useful for applications that receive RTP via **twjit** but don't send out their own originated RTP traffic. To generate a standalone RR packet, call `twna_twrtp_send_rtcp_rr()`. This operation will succeed only if SDES strings have been set, if this **twrtp** instance is equipped with **twjit**, if that **twjit** instance was actually used to receive traffic, and if at least one RTP packet has been received. The content of the generated standalone RR packet is exactly the same as the RR block that is more commonly included in an SR packet, as described in the previous section.

5.4. RTCP support limitations

RTCP support in **twrtp** is subject to the following limitations:

- Sender reports (SR packets) emitted by **twrtp** can only describe traffic that is generated locally via `twna_twrtp_tx_quantum()`, not forwarded traffic that is emitted via `twna_twrtp_tx_forward()`. There is no way to generate SR packets at all outside of `twna_twrtp_tx_quantum()`.
- The library can only generate reception reports (either standalone RR packets or as part of SR packets) for traffic that is received via **twjit**, but not for traffic that is received via non-delayed unbuffered path — see §4.4.
- The built-in RTCP receiver and parser can only extract potentially useful RR info (reports of packet loss and interarrival jitter) from far end reception reports when those far end RRs describe our own Tx SSRC (see §4.5.1), not some foreign SSRC we forward per §4.6.

The summary of these limitations is that **twrtp** has truly functional RTCP support only when **twrtp** is used to implement a full endpoint, one that interfaces between RTP and a fixed timing system such as GSM Um TCH, T1/E1 TDM or a software transcoder that runs on its own CLOCK_MONOTONIC timerfd time base. ‘‘Light’’ RTP endpoints that omit some components of this full endpoint ensemble will most likely be unable to support RTCP.

5.5. Usefulness of RTCP

In the opinion of **twrtp** author, RTCP is most useful in IP-PSTN environment where RTP traffic is exchanged between peer entities under different ownership and different administrative control, traveling across public Internet. In that environment, proper implementation of RTCP can be seen as good netizenship: the administrator of one fiefdom can see **twjit** stats (or full pcap when needed for deeper debugging) on RTP traffic *received* by her queendom, but she can only know if her outgoing traffic suffers from packet loss or jitter if administrators of other fiefdoms have configured *their* systems to emit RTCP reception reports. For this reason, `tw-border-mgw` instances at Themyscira MSCs are configured to dutifully emit RTCP SR (which includes RR block) on IP-PSTN side every 5 s, or after every 250 RTP data packets sent every 20 ms.

On the other hand, RTCP is **not** really useful in a single-administration GSM RAN, i.e., in environments where both ends of the RTP transport leg are controlled by the same administration. In Themyscira environment, each GSM-codec-carrying RTP transport leg runs between `tw-border-mgw` or other ThemWi CN components on one end, located at an MSC site, and either OsmoBTS or `tw-e1abis-mgw` on the other end, located at a cell site, carried across public Internet in a WireGuard tunnel. Because transport across public Internet is involved, RTP performance needs to be closely monitored with an eye out for packet loss, jitter or even reordering, and **twjit** configuration needs to be carefully tuned. However, direct examination of **twjit** stats on both CN and BSS ends will yield much more detailed information than the constrained data model of RTCP — hence RTCP is not really useful.

5.6. Non-RTCP operation

If **twrtp** needs to be used in an environment where RTCP is not needed, or even one where use of RTCP is forbidden, nothing special needs to be done to achieve non-RTCP operation. No RTCP packets will be emitted if the application never calls `twna_twrtp_set_sdes()`; any received RTCP packets will still be parsed as described in §5.2, but the existence of saved bits from this parsing can be simply ignored. RR info from **twjit**, collected as described in §5.1, can be likewise ignored.

6. Stats at twrtp level

In addition to **twjit** stats counters described in Chapter 3, **twrtp** layer has its own stats structure with a few additional counters, dealing with both RTP and RTCP packets in both Rx and Tx directions. This stats structure is `struct twrtp_endp_stats` in `twrtp-proto` version or `struct twna_twrtp_stats` in `twrtp-native`. Here is a description of all counters in this set:

`rx_rtp_pkt`	This counter increments for every packet that is received on the RTP UDP socket **and** has a source address that matches the current remote peer set with `twna_twrtp_set_remote()`.
`rx_rtp_badsrc`	This counter counts packets that were received on the RTP UDP socket, but then discarded because their source address was wrong.
`rx_rtcp_pkt`	This counter increments for every packet that is received on the RTCP UDP socket **and** has a source address that matches the current remote peer set with `twna_twrtp_set_remote()`.
`rx_rtcp_badsrc`	This counter counts packets that were received on the RTCP UDP socket, but then discarded because their source address was wrong.
`rx_rtcp_invalid`	This counter counts packets that were received on the RTCP UDP socket, passed the source address check, but were deemed invalid in parsing.
`rx_rtcp_wrong_ssrc`	This counter increments for every parsed reception report block within a received RTCP SR or RR packet that describes an SSRC other than our Tx SSRC of §4.5.1.
`tx_rtp_pkt`	This counter increments for every RTP data packet emitted via `twna_twrtp_tx_quantum()`; it is also emitted in RTCP SR packets in the

	‘‘sender’s packet count’’ word. Packets transmitted via `twna_twrtp_tx_forward()` are **not** counted here; as explained in §5.4, there is no RTCP support in **twrtp** for this path.
`tx_rtp_bytes`	This counter counts payload bytes transmitted via `twna_twrtp_tx_quantum()`; it is also emitted in RTCP SR packets in the ‘‘sender’s octet count’’ word. Just like `tx_rtp_pkt`, this counter is not affected by `twna_twrtp_tx_forward()` path.
`tx_rtcp_pkt`	This counter increments for every RTCP packet emitted by this **twrtp** instance.