CHANGE REQUEST

DASH-IF IOP	CR		rev	3	Current version:	4.1

Status:		Draft		Internal Review	X	Community Review		Agreed

Title:	Leap second handling clarifications
Source:	DASH-IF IOP
Supporting Companies:	Axiom, Qualcomm Incorporated
Category:	**C** ***Date:*** 2018-01-16

Use one of the following categories:
C *(correction)*
A *(addition of feature)*
B *(editorial modification)*

Reason for change:	Ambiguities with regard to leap second handling were pointed out in https://github.com/Dash-Industry-Forum/DASH-IF-IOP/issues/161
Summary of change:	Clarifies how to handle leap seconds and how to avoid common pitfalls
Consequences if not approved:	DASH packagers and clients may experience noninteroperable behavior during leap seconds

Sections affected:	4.7
Other comments:	

Disclaimer:	This document is not yet final. It is provided for public review until the deadline mentioned below. If you have comments on the document, please submit comments by one of the following means: - at the github repository https://github.com/Dash-IndustryForum/IOP/issues (public at https://gitreports.com/issue/haudiobe/DASH-IF-IOP) - dashif+iop@groupspaces.com with a subject tag [LEAP], or Please add a detailed description of the problem and the comment. Based on the received comments a final document will be published latest by the expected publication date below, integrated in a new version of DASH-IF IOP if the following additional criteria are fulfilled: - All comments from community review are addressed - The relevant aspects for the Conformance Software are provided - Verified IOP test vectors are provided
Commenting Deadline:	March 31st, 2018
Expected Publication:	June 30th, 2018

[Modify] 4.7.2

[Remove page 90 lines 13-15 that talk about leap seconds.]

[New] 4.7.4 Leap Second Handling

A leap second occurs on the UTC timeline every 18 months on average. Leap seconds are regular seconds representing a span of real time, not a mere mathematical trick. There are 61 real seconds in the minute and 3 601 in the hour and 86 401 in the day that contains a leap second.

The MPD availability timeline is calculated on the UTC timeline. Service providers and clients need to be aware of all leap seconds that take place in order to calculate period and segment availability times in an interoperable manner.

Many platforms do not provide built-in time measurement mechanisms that are aware of leap seconds, nor do they provide a list of leap seconds. If no platform-provided leap second aware mechanism for measuring real time is available, one should be synthesized using a public list of leap seconds[x] that is regularly updated.

Table 1. Leap second unaware clocks measure only 1 second of time between 2016-12-31 23:59:59 and 2017-01-01 00:00:00, whereas in reality 2 seconds exist.

True UTC time	*Unix timestamp*	*Timestamp from most "UTC" timekeeping APIs*
2016-12-31 23:59:59	1483228799	2016-12-31 23:59:59
2016-12-31 23:59:60	**1483228800**	**2017-01-01 00:00:00**
2017-01-01 00:00:00	1483228800	2017-01-01 00:00:00

The above table illustrates the primary source of problems in leap second handling – many commonly used APIs, including those based on Unix time and its Microsoft equivalent, are not aware of the existence of leap seconds. Timestamps returned from commonly used APIs either freeze for a second or repeat a second in order to track true UTC time, depending on implementation.

The following are some common use cases where UTC time must be referenced by DASH packagers and clients:

- MPD@availabilityStartTime is a moment on the UTC timeline
- MPD@publishTime is a moment on the UTC timeline
- Clients may schedule segment presentation on the UTC timeline

Timing logic in the MPD often operates on durations of real time that act as offsets from moments on the UTC timeline:

- Period@start is an offset in real time (including leap seconds) from MPD@availabilityStartTime
- Segment availability time is an offset in real time (including leap seconds) from the Period start time
- Segment durations are the span of real time (including leap seconds) that elapses between the availability of consecutive Segments
- Clients may schedule Segment presentation by adding durations of previously presented Segments to an initial moment on the UTC timeline

For accurate timekeeping, all time measurement in packagers and clients should operate in real time and be aware of leap seconds. Specifically, timing logic must:

1. Include leap seconds in any Period and Segment durations calculated by subtracting an end and start timestamp.

2. Perform Segment scheduling either on a leap second aware UTC timeline or on an independent timeline where only the Period is explicitly scheduled and Segments themselves are simply ordered in sequence.

As it is not possible to accurately convert from leap second unaware time to leap second aware time during leap seconds, such conversions should be avoided. For example, instead of calculating Period duration by subtracting the Period start timestamp from the Period end timestamp, a more accurate result may be achieved by adding the durations of all Segments in the Period to the Period start timestamp. The Period start timestamp may in turn equal the (accurately calculated) end timestamp of the previous Period or be fixed to a moment in real time that is known not to be a leap second.

XML processing APIs on platforms that are not leap second aware are generally unable to handle datetime strings that contain a seconds value of 60 (e.g. availabilityStartTime="2016-12-31T23:59:60Z"). Packagers should not write datetime values representing moments in leap seconds into datetime-typed fields in the MPD and should instead round the values up to the nearest non-leap second (e.g. availabilityStartTime="2017-01-01T00:00:00Z" corresponding to above example). If subsecond-accurate timing is desired, rounding of MPD@availabilityStartTime should be avoided by delaying MPD availability so that it does not start within a leap second.

Any timestamps or durations used for Segment scheduling in the MPD must include time spent in leap seconds. The below example illustrates the mapping of template-based and timeline-based Segment scheduling to real time.

Given the following segment template for a period starting at 2016-12-31 23:59:59.000Z, we can calculate the following table for Segment scheduling.

<SegmentTemplate duration=”500” timescale=”1000” startNumber=”1” media=”$Number$.mp4” />

$Number$	Segment timeline timestamp	UTC timestamp	Unix timestamp	“UTC” timestamp as used by common APIs
1	0	2016-12-31 23:59:59.000	1483228799[.0]	2016-12-31 23:59:59.000
2	500	2016-12-31 23:59:59.500	1483228799[.5]	2016-12-31 23:59:59.500
3	**1000**	**2016-12-31 23:59:60.000**	**Cannot represent**	**Cannot represent**
4	**1500**	**2016-12-31 23:59:60.500**	**Cannot represent**	**Cannot represent**
5	2000	2017-01-01 00:00:00.000	1483228800[.0]	2017-01-01 00:00:00.000

As you can see, it is not possible to accurately represent the scheduled moment of Segments 3 and 4 using leap second unaware timekeeping APIs. Packagers and clients must either use a leap second aware clock or only reference such Segments in manner relative to other Segments, without ever calculating an absolute timestamp for them.

Furthermore, any duration calculations based on the leap second unaware timestamps would incorrectly conclude that 1.5 seconds elapsed during playback of the above 2.5 seconds worth of Segments. This error can be overcome by explicitly adding any “lost seconds” to the duration, as long as the leap second unaware start/end timestamps themselves are not within a leap second.

The same principles apply when using time-based segment template or a segment timeline.

[New] 4.7.4.1 Synthesizing a leap second aware clock

On platforms that do not have a leap second aware clock, one can be created based on a list of leap seconds[x] that is combined with a leap second unaware clock that returns Unix timestamps[1].

[1] Other types of clocks can also be substituted but Unix timestamps are used here because the Unix epoch of 1970-01-01 00:00:00 is conveniently before all existing leap seconds.

To calculate the current leap second aware time T_{real} from a Unix timestamp T_{unix} on a common timescale of TS ticks per second, with a list of Unix timestamps L each of which indicates a leap second has passed[2], execute the following algorithm:

1. Calculate the count of elapsed leap seconds as $A = Count(L <= T_{unix})$
2. Calculate the leap second aware timestamp as $T_{real} = T_{unix} + A * TS$

To convert from leap second aware timestamps back to leap second unaware timestamps, execute the following algorithm:

1. Calculate the UTC timestamp of the start of each leap second as $L_n^{realstart} = L_n + Count(L < L_n) * TS$
2. Determine the leap seconds that started before the current timestamp as $L^{active} = L^{realstart} <= T_{real}$
3. Determine how much of any ongoing leap second has not yet elapsed as $D_{remaining} = Max(0, Max(L^{active}) + TS - T_{real})$
4. Calculate the leap second unaware Unix timestamp as $T_{unix} = T_{real} - Count(L^{active}) + D_{remaining}$

This synthetic clock returns timestamps that include elapsed leap seconds, thereby ensuring measuring duration $D = T_{end} – T_{start}$ includes any time spent in leap seconds.

While the synthetic clock can account for leap seconds in arithmetic, it cannot return real time values that are themselves within a leap second. This results in the following inaccuracy:

1. With a 1 tick per second timescale, time stands still for 1 second during a leap second, after which it jumps past the leap second, never entering it.
2. With a finer timescale (e.g. 1000 ticks per second), the moments in the second immediately after a leap second elapse twice, once during the leap second and once after the leap second. In other words, there occurs a moment where the clock rolls back by one second.

To prevent the above inaccuracy from affecting calculations, implementations should, whenever possible, measure time by adding elapsed segment durations (which are in real time and thus include time spent in leap seconds) to an initial real-time timestamp returned from the synthetic clock.

Converting leap second aware timestamps back to leap second unaware timestamps is inaccurate if the former is currently within a leap second. In such a case, the value is rounded up to the nearest non-leap second.

[New references]

[x] . https://www.ietf.org/timezones/data/leap-seconds.list

---

[2] Note that the list of leap seconds [x] needs processing in order to transform it to the suitable form, as it does not use the Unix epoch for timestamps and the adjustment values also include an additional 10-second offset between TAI and UTC. For example, the timestamp 2303683200 in [x] references 1973-01-01 00:00:00 UTC which is 94694400 seconds from the Unix epoch and the corresponding adjustment value 12 in [x] means that there are 2 leap seconds between this moment and the Unix epoch (after subtracting the 10 second TAI-UTC offset).