GTFS-RT Differential Semantics

TL;DR

The objective of this document is to establish how a series of differential GTFS-RT messages may be combined to provide the feed consumer an identical result to fetching the full dataset at once, but using less bandwidth and providing more immediate updates.

In short, we recommend that the word "differential" be understood to mean incremental reception of a GTFS-RT update dataset one trip/vehicle at a time, rather than the accumulation of successive messages about the same trip/vehicle.

The producer/consumer relationship should be stateless, and all GTFS-RT messages except Alerts should be idempotent: each message is applied against the relevant trip in the scheduled GTFS feed, with no consideration for any messages previously received about the trip/vehicle.

Introduction

Developers working on real-time passenger information and routing in the Netherlands have implemented differential (also called “incremental”) GTFS-RT out of both necessity and technical interest in realizing this part of the specification. To our knowledge the differential mode has not been used anywhere in production, so new rules must be established. We have been working on producing and consuming such data for about 1.5 years now, and we would like to offer some recommendations for the semantics of differential mode based on our experience.

The GTFS-RT protocol buffer specification contains a field called “Incrementality” in the FeedHeader message type. The accompanying comment states that it “Determines whether the current fetch is incremental. Currently, DIFFERENTIAL mode is unsupported and behavior is unspecified for feeds that use this mode. There are discussions on the GTFS-realtime mailing list around fully specifying the behavior of DIFFERENTIAL mode and the documentation will be updated when those discussions are finalized.” We believe that based on existing work, it is now possible to formalize the meaning and structure of differential GFTS-RT.

Context: The Netherlands

The proportion of public transit data available under open licenses in the Netherlands is nearly 100 percent thanks to unique legal and historical circumstances as well as consistent pressure and implementation work by open data advocacy organizations. This is true not only of static schedule data but also of real time data including schedule deviations, disruptions, and cancellations. Operators are required to provide the position and schedule deviation of every vehicle at least once per minute to any national-scale transport information provider, and

OpenOV is integrating and redistributing this data in GTFS-RT format under a completely open license. See *http://ovradar.nl* for a real-time visualization of this data source.

The high update frequency and sheer volume of data makes streaming incremental delivery of GTFS-RT an appealing option. When using such large, national-scale feeds in the more common FULL_DATASET mode, the consumer may be forced to poll the producer's server rather infrequently to avoid excessive load on both the producer and the consumer. Repeatedly transmitting and interpreting redundant, unchanged information is potentially wasteful of both bandwidth and processor time. In addition, full-dataset polling compromises the immediacy of the updates. In a polling-based system, each component (the in-vehicle device, arrival prediction, integration, the consumer) has a characteristic update frequency which may be measured in minutes, and the cascading delay may be deemed unacceptable. These problems can be eliminated by a message-passing system and differential GTFS-RT.

Recommendations

1. Message Transport

The GTFS-RT specification consists largely of a protocol buffer definition and accompanying documentation at *https://developers.google.com/transit/gtfs-realtime/reference*. This reference states that the protocol buffer messages are retrieved via HTTP GET. To allow flexible and effective use of differential mode, the GTFS-RT specification should instead allow messages to reach the consumer by any protocol capable of delivering them in one piece.

Early Dutch experiments with OpenTripPlanner and GTFS-RT used ZeroMQ as a message-oriented transport layer. Brian Ferris of Google Transit and One Bus Away advocated the use of websockets in an informal discussion. We thought it wise to support the emergence of standardized bidirectional message-oriented communication and websockets have been implemented in all major browsers and web servers, so we migrated to websockets shortly thereafter. The only serious problems we have run into when distributing GTFS-RT this way are concurrency bugs in websocket libraries.

While it should not be required to distribute GTFS-RT over a particular protocol, currently the *de facto* standard way to deliver GTFS-RT protocol buffer messages is by HTTP (for full data sets) and websockets (for differential mode). Producers should use these protocols unless there is some compelling technical reason to do otherwise.

Note that websockets are implemented as an HTTP request with an Upgrade header. Therefore, in practice the same URL may be used for both full datasets (HTTP GET) and message-oriented differential feeds (websockets).

In fact, the push/pull distinction is orthogonal to the FULL_DATASET/DIFFERENTIAL distinction, and all four combinations are possible. Below are concrete examples.

FULL_DATASET producer push: The producer uses HTTP POST to send the consumer a document containing all currently active Alerts whenever an Alert is added or removed.

FULL_DATASET consumer pull: The consumer polls the producer at regular intervals, using HTTP GET to retrieve a document containing all currently relevant VehiclePositions.

DIFFERENTIAL producer push: The consumer opens a persistent websocket connection to the producer. The producer sends individual TripUpdate messages containing arrival time predictions as vehicles report in.

DIFFERENTIAL consumer pull: This would require the producer to maintain per-consumer state or employ accurate timestamps, and we currently see no practical use case.

2. "Priming" consumer state

Our goal is to make sure that a consumer can combine a series of differential messages to achieve an identical result to fetching the full dataset at once. There are two options for ensuring that the client has a completely reconstructed copy of the real-time state:

A. Provide one update per active vehicle at a guaranteed frequency. After a certain delay, the client should possess a full real-time dataset.

B. Send the full dataset upon establishing the connection, then switch to incremental updates. The consumer is assured of having a coherent, current copy of the real-time dataset within moments of connecting.

We have adopted the second approach, though thanks to high update frequencies the first option would also work for Dutch data.

3. Update granularity and accumulation

A single top-level FeedMessage may contain any number of TripUpdates, Alerts, or VehiclePositions. A series of differential FeedMessages may contain complementary or contradictory information about the same GTFS entity (trip or vehicle). We need to establish how the effects of a series of differential FeedMessages accumulate.

Our recommendation is that each incoming GTFS-RT TripUpdate or VehiclePosition be applied directly against the specified entity in the original GTFS schedule feed, with no regard at all for previous GTFS-RT messages concerning the same entity. A FeedMessage should not contain more than one message that applies to the same GTFS entity, and when it does the effect is undefined. In the interest of tolerating malformed input, the consumer should apply only one of the messages and ignore the others.

An apparent difficulty arises when trips which do not exist in the original GTFS feed are added. It must be possible to cancel or update these trips which only exist due to previous GTFS-RT messages. The solution is actually relatively simple: Messages about GTFS entities that were

not included in the original schedule feed are, in a sense, applied “against their absence” in the original feed. Each such message is interpreted on a blank slate as if no other messages for that same entity ID have ever been received. Thus, the same logic can be applied every time a message is received for a given trip or vehicle.

Finally, the consumer should ensure that updates received together in a single FeedMessage become visible to an end user simultaneously. The user should never observe a state where some updates in a FeedMessage have been applied but others have not. This allows *e.g.* replacing a scheduled trip with a longer added one via a combination of CANCELLED and ADDED TripUpdates.

4. GTFS-RT message deletion

The GTFS-RT reference document states that the *FeedEntity.id* and *FeedEntity.isDeleted* fields are only relevant to incremental feeds. However, given the differential semantics outlined above there is not a clear need to explicitly delete RT messages (other than Alerts, see discussion below) as one can always simply provide an update that will supersede them. Explicit deletion seems to carry a higher risk of desynchronization between producer and consumer views of the real-time update dataset, and imposes additional message ID bookkeeping on both sides.

Depending on deletion semantics, update deletion can also force the consumer to retain an unbounded number of RT messages. Consider the case where a large number of updates have been sent for the same trip, then deletion messages are sent for each of those updates in reverse order. Though individual messages are interpreted against the original schedule feed alone, the consumer must maintain a list of all those past messages “just in case” one of them needs to be re-applied to return to a previous state.

Some examples of how messages for each update type can be effectively (rather than explicitly) deleted:

TripUpdates. Both scheduled and added trips can be CANCELED by sending another TripUpdate message referencing the same trip ID.

VehiclePositions. When a message is received saying the vehicle is STOPPED_AT the last stop in a trip, or saying that a vehicle has moved on to a different trip, its state with respect to that first trip is effectively removed. However, it might be helpful to provide a means to unambiguously signal that a vehicle has gone out of service.

Alerts. Unlike the other two message types, multiple Alerts may accumulate to the same GTFS entity. Without alert IDs and deletion flags, it would not be possible to remove them in differential mode. We can completely sidestep the problem by simply not using differential Alerts. Alert datasets are generally small and do not benefit greatly from low update latency. They seem to be a good use case for the FULL_DATASET provider push combination.

It is arguably awkward to disallow differential alerts, and RT message IDs can be practical for managing message flows in GTFS-RT producers. If update IDs are retained, another solution to the above deletion problems fon non-Alert entities is A) only allow deleting the most recent update for a given trip/vehicle via RT update IDs, and B) define deletion as returning the consumer to the initial (blank-slate / scheduled GTFS feed) state for that particular entity. This is the approach currently assumed by the openOV integration tools.

5. Strict vs. Lenient feeds

In a conversation between PlannerStack, OpenOV, and Google Transit representatives about GTFS-RT, the idea of a strict vs. lenient feed distinction arose. The GTFS-RT spec allows TripUpdates which contain StopTimeUpdates for only a subset of the stops defined in the original scheduled feed. These TripUpdates may even provide StopTimeUpdates for non-contiguous sets of stops, forcing the consumer to propagate delays through intermediate stops and apply prediction algorithms.

The GTFS-RT spec allows for partial or sparse TripUpdates because many agencies are not able to provide accurate downstream arrival times for a vehicle. In the absence of reliable prediction software, they prefer to provide only what they know with some level of certainty, *e.g.* the current delay of an individual trip relative to its original schedule.

While this does make sense when no more detailed information is available, consider the case where such incomplete TripUpdates are used to add unscheduled trips. The feed producer may introduce backtracking, loops, reversals, skipped stops, etc. all using an abbreviated form where only a subset of the trip's stops are mentioned. A rather large number of rules is needed to enable a sort of shorthand that may provide little practical advantage over full, detailed updates.

The proposed definition of a strict feed is quite simple: every TripUpdate must include a StopTimeUpdate for every stop in the entire trip, with no exceptions. This would apply to both messages updating scheduled trips and messages adding new unscheduled trips. A strict feed is one that leaves consumers no room at all for interpretation.

OpenTripPlanner and Bliksem's RRRR (and presumably any other journey planning system which applies realtime updates to routing) must merge real-time updates with the original schedule for a given trip, smoothing over any ambiguities and missing information to produce a complete series of reported, projected, or propagated arrival and departure times for the entire length of the trip.

A feed that does not conform to the "strict" convention for communicating delays forces the consumer to perform arrival time prediction and introduces a risk that various passenger information products will provide different predictions, perhaps deviating from the predictions used by the feed producing organization itself. This may in fact be the desired result, but when delivering such incomplete information a feed producer would need to set *strict=false*,

acknowledging the fact that they are providing feed consumers with incomplete information and that end users will be depending on third-party heuristics.

One advantage of defining a strict mode is that it clearly delineates the role of a GTFS-RT “inference” module (the name is perhaps not ideal). Inference combines non-strict GTFS-RT with its corresponding GTFS schedule feed and produces strict GTFS-RT, which is unambiguous and usable directly for routing. Such a module could do anything from following the most basic delay propagation rules to applying proprietary arrival time prediction methods, but its role in the chain is well-defined.

It is perhaps valuable to distinguish between strict updates to scheduled trips and strict added trips. The former greatly reduces the number of rules needed for the feed consumer to merge the update message with the scheduled trip, while the latter only avoids delay prediction and propagation by the feed consumer.

Additional Concerns

Some of the comments below suggest changes that are not backward compatible. It is not clear whether this is a problem since the GTFS-RT spec already includes version numbers, and to our knowledge the incremental RT fields are not yet widely used.

1. **Differential vs. Incremental.** The reference and specification use both terms: differential and incremental. We should choose only one and use it consistently. These same terms are used in backup systems, and the behavior here seems closer to incremental backups where only individual files that have changed since the last backup (be it incremental, differential, or full) are included. In some sense, GTFS-RT as a whole is “differential” with respect to scheduled GTFS, and the new mode we are discussing is “incremental” with respect to a GTFS-RT FULL_DATASET.

2. **Incrementality enum.** It seems more natural to make this a boolean field called “incremental”. In the existing spec, this field introduces three terms (incrementality, differential, full_dataset) to express a single predicate.

3. **ScheduleRelationship enums**. There are two different enums with the same name, one inside StopTimeUpdate and one inside TripDescripton. Also, the definition of UNSCHEDULED is not clear. How is it different than ADDED? It is possible to have a TripUpdate with ScheduleRelationship=ADDED, but all its stops are ScheduleRelationship=SCHEDULED. This is strange.

4. **TripDescriptors have multiple roles**. They serve to choose an entity to act upon, but also act upon that entity in the case of DELETED.

5. **Feeds can contain only one message type.** GTFS-RT includes three message types: TripUpdates, ServiceAlerts, and VehiclePositions. The reference document states that "Even though the gtfs-realtime.proto syntax allows multiple entity types to be mixed for a feed, only one type of entity can be used in a particular feed." First, why is this restriction imposed? In the case of differential updates, what constitutes a feed? (A single, specific URL, or a single transport-layer message, or a single FeedMessage?)

6. **Google-specific details in the GTFS-RT reference**. The reference states that "A feed depends on some external configuration" including the entity type and the polling frequency. There are no min_update_delay and max_update_delay fields mentioned elsewhere in the reference, perhaps these refer to some Google Transit configuration.

7. **HTTP is required.** The reference states that "Each message in the stream is obtained as a response to an appropriate HTTP GET request." Normative references to the protocol carrying GTFS-RT messages should be removed. The meaning of the words "stream" and "message" is not clear.

8. **Streaming vs. chunked.** Besides the incremental vs. full-dataset distinction, a streaming vs. chunked distinction could be useful in discussions. There is actually a continuum of incremental GTFS-RT, from FeedMessages that are very small (a FeedEntity containing a TripUpdate for a single trip), up to FeedMessages containing entities accumulated over several seconds or minutes. Highly chunked incremental GTFS-RT begins to resemble a full dataset.

9. **Problems with added trips.** Consider the case where a new, unscheduled trip is added, with all stops specified in the initial TripUpdate message. Our blank-slate semantics make it impossible to subsequently provide a partial set of stoptimes or a delay figure for that trip. While this is convenient from the consumer perspective, it requires the producer to always emit “strict” TripUpdates for added trips.

Footnote on the need for some indicator (perhaps in ScheduleRelationship) that a stop has been passed: The FeedHeader contains a timestamp, and stoptimes can even be compared with the system clock, so there is no need for the equivalent of KV8 PASSED. Besides, this duplicates functionality in VehiclePosition messages.