Points of interest

Contents

Use-cases

General points of interest

Third-party applications (the "provider") might be aware of general "points of interest" for mapping. For example, an accommodation booking app-bundle might be aware of the locations of hotels, either from a particular chain if the app-bundle is for that chain, or for all hotels if the app-bundle is for a general service like Trip Advisor.

- It must be possible for a third-party app-bundle to provide these "points of interest" to a navigation app.
- It must be possible for the third-party app-bundle to contain a non-GUI service (an "agent") which will act as the POI provider.
- The navigation app should not be flooded with irrelevant points of interest, for example a hotel in Cambridge while driving from Hannover to Hildesheim
- For privacy, if an application bundle advertises that it is a points-of-interest provider but its manifest indicates that it is not intended to have access to location data, the navigation app must not reveal the current location to it.
- If the POI provider is implemented in terms of an Internet service that can provide a filter/search-based stream of points of interest, it must be possible for the POI provider to avoid being flooded with irrelevant points of interest by the remote server.
- This pattern could potentially be generalized to replace the navigation app with any other consumer.
- We will assume that there is no particular preferred encoding that will be

used by a majority of applications, so re-encoding into a common format will usually be necessary.

TPEG

One specific instance of General points of interest is receiving TPEG[1] broadcasts. These are typically carried on digital radio (DAB) and can encode time- and location-bound events such as traffic congestion, road closures and weather. They can also be transferred via the Internet.

When TPEG is broadcast over DAB, this is done with a "carousel[2]" approach in which the transmitter repeatedly cycles through a list of currently-valid messages. It is not currently clear to us whether TPEG feeds over the Internet follow the same "carousel" design.

It is not currently clear to us whether TPEG feeds over the Internet filter for relevant events at the client- or server-side.

- It must be possible for an app-bundle to contain an agent that receives TPEG broadcasts and provides events to a navigation app.
- The navigation app should not be flooded with irrelevant points of interest, for example traffic congestion that is not close to either the current location or the projected route.
  - If filtering is performed at the server side, the TPEG provider agent should be able to give the server a suitable filter.
  - If filtering is performed at the client side, it could take place in the agent rather than the navigation app, to minimize IPC traffic between them.
  - If filtering is performed in the navigation app itself, the channel between the TPEG provider agent and the navigation app must be one that is suitable for high-throughput events, and should be able to feed back that events are being transferred faster than the navigation app can read them, so that the TPEG provider agent can "back off" by omitting some events.
- More than one TPEG provider can be installed at the same time. If they are, they are all used in parallel.
- A brochure from the Institut für Rundfunktechnik[3] uses 64 kbit/s = 56 messages/s as an indicative figure for TPEG over DAB. This implies that an average binary message should be in the range 100-200 bytes.
- The European Broadcasting Union's document "TPEG - What is it all about?"[4] suggests that TPEG-over-Internet clients are expected to poll servers, using a model in which the client downloads a comparatively large

[1] http://tisa.org/technologies/tpeg/
[2] https://en.wikipedia.org/wiki/Data_and_object_carousel
[3] https://www.easyway-its.eu/sites/default/files/EW-Highlight_Bavaria_Multimodal_TPEG.pdf
[4] https://tech.ebu.ch/docs/other/TPEG-what-is-it.pdf

"initial state" at startup, and subsequently receives "deltas" from that initial state.

Specific points of interest

Third-party applications (the "provider") might be aware of "points of interest" that are relevant to the user with a high probability. For example, an accommodation or travel booking app-bundle might be aware that the user has booked a stay at a particular hotel, or a flight from a particular airport; or a PIM application might be aware of the location of the user's home or normal place of work, or an upcoming appointment.

- It must be possible for a third-party app-bundle to provide these "points of interest" to a navigation app.
- Unlike General points of interest, being flooded with irrelevant points of interest is unlikely to be a problem here, because the "provider" application knows that these points of interest are highly likely to be relevant.
- Time-based filtering could be useful here: appointments beyond a defined date range might not be considered relevant.
- This pattern could potentially be generalized to replace the navigation app with a non-specific "sink".

Weather

Weather information might be queried from an Internet service or decoded from TPEG broadcasts by an agent in a built-in, preinstalled or third-party app-bundle.

- The requirements closely resemble General points of interest: we require the weather at one or a few weather stations closest to our location or projected route, and we are not interested in distant weather reports.
- Date-based queries (e.g. projected weather for the next week while planning a long drive) might also be useful here.
- We anticipate that the data rate here will be considerably lower than for TPEG in general.

Security and privacy considerations

Access to location information

As noted above, if an application bundle advertises that it is a points-of-interest provider but its manifest indicates that it is not intended to have access to location data, the navigation app must not reveal the current location to it.

Attack surface

TPEG messages come from DAB broadcasts or the Internet.

Regardless of how well we might mitigate attacks, if an attacker is able to send crafted TPEG messages to the Apertis system in a way that is indistinguishable from legitimate data, then that attacker can cause the navigation app to display points-of-interest of their choice. This is unavoidable, and is mentioned here only for completeness.

Beyond that, if attacker can cause the Apertis system to receive crafted TPEG messages, they might be able to exploit implementation errors in a TPEG parser. We consider three threat-models here:

- assume that the parser is robust and will not crash or misbehave with malicious input;
- alternatively, assume that the parser has a denial-of-service vulnerability which causes it to crash or otherwise cease to process information, but does not allow arbitrary code execution;
- alternatively, assume that the parser has an arbitrary code execution vulnerability which causes it to execute attacker-chosen code

The second and third threat models should be considered to be vulnerabilities (security-sensitive bugs) in the TPEG-parsing component; and can be made less likely by using techniques such as fuzz testing[5] to verify the robustness of the parser. However, it might be considered valuable to mitigate any vulnerabilties in those classes that remain and are discovered by an attacker.

Similarly, a non-TPEG-based points-of-interest or weather information provider is likely to receive data in some non-TPEG format from the Internet, and equivalent security considerations apply to that data. We recommend that cryptographically protected channels such as HTTPS are used where possible.

Other requirements

It is undesirable to increase coupling between consumers and providers by having consumers provide any location/route information to providers. Instead, providers should retrieve that information from the platform.

Similarly, it would be possible for the consumer to give the provider an indication of what is required (in terms of level of detail, search radius around the location and so on). This would allow the provider to optimize its trade-off between resource consumption and information provided, by providing what is needed but no more. However, this causes relatively tight coupling between consumers and providers, which is undesired in an app-centric model. This should not be implemented; instead, the provider should be responsible for determining a reasonable policy.

[5] https://en.wikipedia.org/wiki/Fuzz_testing

Open questions

- Does TPEG already have a defined encoding for points-of-interest similar to those discussed in General points of interest and Specific points of interest, or does it only have encodings for traffic, weather information, and a few specific classes of point-of-interest such as parking?
- Are the data rates discussed above (64 kbit/sec, 56 messages/sec) representative?
- How frequently do we anticipate that a consumer would wish to consume some families of location-sensitive data ("applications" in TPEG jargon - traffic information, weather, points of interest) but not others?
- Do we expect weather information to come from TPEG, or from a query/lookup-based web service with requests like "tell me the weather in Hildesheim tomorrow", or a combination of the two?

Recommendations

See Data sharing[6] for background information on various possible communication between apps. The *consumer* here is the navigation app, and the *providers* are the POI, TPEG and weather providers.

We anticipate that the navigation app (or other consumer) might either be a HMI that is started via user action or a platform component that is started automatically on boot, whereas the various providers will be agents provided by platform components and/or store applications, started either on boot or on-demand. Because the consumer might be a HMI, we recommend that it should act as the *initiator* as described in Data sharing[7].

We recommend that the navigation app should locate suitable POI, weather and/or TPEG providers by performing interface discovery[8].

To bypass concerns about Access to location information and ensure that consumers and providers remain "loosely coupled", we recommend that the consumer does not inform providers about the current location or route. Instead, each provider that requires this information should retrieve it from a platform service via a D-Bus API, following the conventional publish/subscribe model. The platform should only allow this access for providers whose app-bundle manifests authorize it.

TPEG

For TPEG, there is a choice between several solutions. The answers to the Open questions above influence the choice between these options.

[6] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/
[7] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/
[8] https://martyn.pages.apertis.org/apertis-website/concepts/interface_discovery/

Our provisional recommendation is to take the TPEG stream design, and minimize exploitable bugs in TPEG parsing by subjecting the parser to fuzz testing and code auditing.

Where TPEG is received with a carousel[9] model, we recommend that the TPEG provider is responsible for keeping a cache of items received during previous sessions, forgetting those items when they are no longer valid or relevant, and replaying them when each new consumer connects.

TPEG stream

One option is to use data sharing#Consumer-initiated push via a stream[10]. In this model, the method that is called by the provider would start a stream of TPEG messages, in either the binary or XML encoding, with a suitable framing protocol (which will need to be specified) if one is required.

This solution has the advantage that it allows a TPEG provider to treat TPEG as opaque: for example, a DAB radio receiver that is primarily designed to play audio, but receives TPEG via DAB as a side-effect, could pass TPEG messages through to navigation software without parsing or understanding them. This avoids requiring TPEG parsing in DAB applications or agents, leading to high throughput and low resource consumption by these processes (but potentially a correspondingly higher resource consumption for the consumer, which does need to parse the TPEG).

The disadvantage of this solution is that if a provider takes this TPEG pass-through approach, the consumer is directly exposed to potentially malicious data received from the network; for example, a malformed TPEG message might be designed to exploit bugs in the consumer's parser.

Stream of some other format

Another option is to use data sharing#Consumer-initiated push via a stream[11], but require the messages carried by the stream to be in some other format instead of TPEG, for example KML[12]. This would force the provider to parse TPEG and re-encode it in the other format, which comes with an efficiency cost. It also requires the choice and implementation of the other format; if a new format is designed, requirements-gathering and design will be needed, which could be viewed as a waste of effort that could be avoided by reusing an existing format.

All the usual trade-offs between data formats - size efficiency, time efficiency, expressiveness, scope for debugging, and so on - apply to the choice of the other

[9] https://en.wikipedia.org/wiki/Data_and_object_carousel

[10] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/#consumer-initiated-push-via-a-stream

[11] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/#consumer-initiated-push-via-a-stream

[12] https://en.wikipedia.org/wiki/Keyhole_Markup_Language

format. GVariant, JSON or XML might make a reasonable basis for a data format, but each of those would require a suitable data representation (schema) to be designed and specified, similar to the way KML is layered above XML.

If the TPEG parser is assumed to be robust, this approach has no advantage over the TPEG stream. The advantage of this option over a TPEG stream is that if the TPEG parser is not considered to be robust, this option somewhat reduces the attack surface available to potentially-malicious TPEG messages.

If the attacker is only assumed to be able to cause a crash via a malformed message, this is entirely mitigated by performing TPEG parsing in the provider: the consumer would stop receiving messages from that provider, but continue to run. Optionally, it could detect the unexpected end-of-stream and re-initiate communication with a new instance of the provider.

However, if the attacker is assumed to be able to cause arbitrary code execution in the TPEG parser, this design does not necessarily provide any mitigation: having subverted the TPEG provider, the attacker could send arbitrary messages to the consumer in the other format, potentially triggering denial-of-service or code-execution vulnerabilities in the parser for that other format. This could be mitigated by requiring that the parser for the other format has a robust and well-understood implementation provided by the platform.

Publish/subscribe via D-Bus

TPEG providers could use a publish/subscribe approach via D-Bus[13], with each D-Bus message either carrying a binary or XML TPEG message, or a message translated into a non-TPEG encoding using D-Bus data structures. However, our current understanding of the expected message rate is that it is at the high end of the rates for which D-Bus was designed, so we do not recommend this.

In addition, if the TPEG provider does not cache currently-valid TPEG messages in memory but merely passes them through as they are received, then this would be a poor fit for conventional D-Bus design patterns, since a GetCurrentState() method call would not make sense (the provider would not have any current state to get).

General POI providers

For POI providers that are implemented in terms of access to a web service, either the POI provider can perform a series of search operations on the remote server and get relevant POIs in response (a "pull" model), or the remote server can send a stream of POIs to the POI provider while periodically polling the current locations of interest (a "push" model).

For the transport between the POI provider and the POI consumer, multiple approaches are possible, depending on the answers to the Open questions.

[13] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/#publish.2fsubscribe-via-d-bus

Our provisional recommendation is that points of interest are encoded as TPEG and transferred to consumers via the same mechanisms we would use for a TPEG stream.

TPEG stream

If TPEG has an encoding for points of interest, then the same approach can be taken as for TPEG: data sharing#Consumer-initiated push via a stream[14], with a stream of binary or XML TPEG messages. This has the advantage that we do not need to define our own encoding for points of interest.

Another advantage of this approach (assuming that TPEG providers also provide a TPEG stream) is that the consumer only needs to consume TPEG, rather than consuming both TPEG (from DAB broadcasts) and some other format (from general POI providers). If TPEG does not already have an encoding for points of interest, but it is feasible to add one, it might still be worthwhile to do so in order to receive the second advantage.

The disadvantage is that each provider needs to encode whatever information it provides into TPEG format. This could introduce a loss of information if TPEG is not sufficiently expressive to describe everything the provider requires.

If the corresponding solution is not chosen for TPEG providers, this approach should not be chosen either.

Stream of some other format

As with TPEG, we could define a non-TPEG encoding for points of interest and stream them to the consumer using Data sharing#Consumer-initiated push via a stream[15]. Similar considerations apply to the design of the new format: we could select an existing format such as KML[16], or design a new one, perhaps using GVariant or JSON as a base.

If the non-TPEG encoding is chosen well, it should be possible to encode all aspects of a point of interest without information loss. However, if the non-TPEG encoding is not sufficiently expressive, information might still be lost in translation.

Publish/subscribe via D-Bus

As with TPEG, POI providers could use a publish/subscribe approach via D-Bus[17], with each D-Bus message carrying one or more points of interest, either

[14] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/#consumer-initiated-push-via-a-stream

[15] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/#consumer-initiated-push-via-a-stream

[16] https://en.wikipedia.org/wiki/Keyhole_Markup_Language

[17] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/#publish.2fsubscribe-via-d-bus

described using D-Bus data structures or as an opaque byte-array in some known format. We anticipate that POI providers would normally have rather lower message rates than TPEG, so they might be within the range for which D-Bus is designed, even if TPEG is not.

Specific POI providers

The implementation options for specific POI providers are essentially the same as general POI providers, although their requirements are less stringent: the message rate is anticipated to be lower, and location information is not necessarily needed.

We recommend that the same interface as for general points of interest is used: we do not see any reason why we need to distinguish between the two.

Weather

One possible implementation is to use the same TPEG-stream-based design as the other use cases. This is appropriate if we anticipate that the majority of providers of weather information will be implemented using TPEG or similar, but it is a poor fit for a more query-based provider such as one that accesses a web service: the provider would have to perform speculative queries covering the area around the location and/or route, and emit their results at intervals as TPEG or TPEG-like messages.

Another possible implementation is to use Data_sharing#Query-based access via D-Bus[18], which is a good fit for implementations that query a web service, but a poor fit for implementations that receive weather from TPEG (which would have to cache all available weather information, and use their cache to be able to answer queries). This could be mitigated by treating weather queries as a platform service, so that the cache is maintained by the platform.

A third possibility is to have two separate weather APIs - one stream-based design for "ambient weather information", and one query-based design for "current and forecasted weather queries" - and require consumers to choose whether to use one or both depending on their particular requirements. This design has the advantage that each of those APIs represents the underlying weather service in the most natural way, but has the disadvantage that consumers are expected to use two parallel APIs.

We do not currently have a specific recommendation here, and would appreciate feedback on the relative priorities of those designs' strengths and weaknesses.

[18] https://martyn.pages.apertis.org/apertis-website/architecture/data_sharing/#query-based-access-via-d-bus

Dealing with multiple categories

This recommendation is conditional on the answers to the Open questions, above.

We recommend using a name such as "categories", "types" or "classes" for TPEG's jargon term "applications", to avoid confusion with the already overloaded term "application". Documentation could mention the TPEG jargon for clarification, for example:

```
/** * ... * Return a list of categories of location information that are supported * by this provider. These categories are the same concept as * "applications" in TPEG terminology. * ... */ gchar **..._list_categories (... *self);
```

If we anticipate that all categories of data will typically all have the same consumers, we recommend having a single shared interface for interface discovery[19], perhaps `org.apertis.LocationInfoProvider` or `org.apertis.TPEGProvider`. Clients not requiring all of the available data for that interface could receive and discard it.

If we anticipate that categories will often have different consumers, we will need a list of categories, and an interface for interface discovery[20] per category, for example `PointsOfInterestProvider`, `TrafficProvider` and `WeatherProvider`.

One approach to these categories would be to define a separate D-Bus interface per interface-discovery interface, with intentionally similar D-Bus APIs to set up a stream. Each provider that could provide more than one category would be required to demultiplex the messages that it receives and write them into the appropriate streams.

Another approach to these categories would be to define a separate D-Bus interface per interface-discovery interface, with intentionally similar D-Bus APIs to set up a stream. Each provider that could provide more than one category would be required to demultiplex the messages that it receives and write them into the appropriate streams.

[19] https://martyn.pages.apertis.org/apertis-website/concepts/interface_discovery/
[20] https://martyn.pages.apertis.org/apertis-website/concepts/interface_discovery/