APERTIS

Geolocation and navigation

Contents

This documents existing solutions for geo-related features (sometimes known as location-based services, LBS) which could be integrated into Apertis for providing geolocation, geofencing, geocoding and navigation routing support for application bundles.

As of version 0.3.0, the recommended solutions for most of the geo-requirements for Apertis are already implemented as open source libraries which can be integrated into Apertis. Some of them require upstream work to add smaller missing features.

Larger pieces of work need to be done to add address completion and geofencing features to existing open source projects.

The major considerations with all of these features are:

- Whether the feature needs to work offline or can require the vehicle to have an internet connection.
- Privacy sensitivity of the data used or made available by the feature — for example, access to geolocation or navigation routing data is privacy sensitive as it gives the user's location.

- All features must support pluggable backends to allow proprietary solutions to be used if provided by the automotive domain.

The scope of this design is restricted to providing services to applications which need to handle locations or location-based data. This design does not aim to provide APIs suitable for implementing a full vehicle navigation routing system —this is assumed to be provided by the automotive domain[1], and may even provide some of the implementations of geo-related features used by other applications. This means that the navigation routing API suggested by this design is limited to allowing applications to interact with an external navigation routing system, rather than implement or embed one themselves. Appendix: Recommendations for third-party navigation applications when implementing a navigation application are given.

Terminology and concepts

Coordinates

Throughout this document, *coordinates* (or *a coordinate pair*) are taken to mean a latitude and longitude describing a single point in some well-defined coordinate system (typically WGS84).

Geolocation

Geolocation is the resolution of the vehicle's current location to a coordinate pair. It might not be possible to geolocate at any given time, due to unavailability of sensor input such as a GPS lock.

Forward geocoding

Forward geocoding is the parsing of an address or textual description of a location, and returning zero or more coordinates which match it.

Reverse geocoding

Reverse geocoding is the lookup of the zero or one addresses which correspond to a coordinate pair.

Geofencing

Geofencing is a system for notifying application bundles when the vehicle enters a pre-defined 'fenced'area. For example, this can be used for notifying about jobs to do in a particular area the vehicle is passing through, or for detecting the end of a navigation route.

[1] https://www.apertis.org/glossary/#automotive-domain

Route planning

Route planning is where a start, destination and zero or more via-points are specified by the user, and the system plans a road navigation route between them, potentially optimising for traffic conditions or route length.

Route replanning

Route replanning is where a route is recalculated to follow different roads, without changing the start, destination or any via-points along the way. This could happen if the driver took a wrong turn, or if traffic conditions change, for example.

Route cancellation

Route cancellation is when a route in progress has its destination or via-points changed or removed. This does not necessarily happen when the vehicle is stopped or the ignition turned off, as route navigation could continue after an over-night stop, for example.

Point of interest

A *point of interest* (*POI*) is a specific location which someone (a driver or passenger) might find interesting, such as a hotel, restaurant, fuel station or tourist attraction.

Route list

A *route list* is the geometry of a navigation route, including the start point, all destinations and all vertices and edges which unambiguously describe the set of roads the route should use. Note that it is different from *route guidance*, which is the set of instructions to follow for the route.

Horizon

The *horizon* is the collection of all interesting objects which are ahead of the driver on their route ('on the horizon'). Practically, this is a combination of upcoming *points of interest*, and the remaining *route list*.

Route guidance

Route guidance is the set of turn-by-turn instructions for following a navigation route, such as 'take the next left'or 'continue along the A14 for 57km'. It is not the *route list*, which is the geometry of the route, but it may be possible to derive it from the route list.

Text-to-speech (TTS)

Text-to-speech (TTS) is a user interface technology for outputting a user interface as computer generated speech.

Location-based services (LBS)

Location-based services (LBS) is another name for the collection of geo-related features provided to applications: geolocation, geofencing, geocoding and navigation routing.

Navigation application

A *navigation application* is assumed (for the purposes of this document) to be an application bundle which contains

- a *navigation UI* for choosing a destination and planning a route;
- a *guidance UI* for providing guidance for that route while driving, potentially also showing points of interest along the way;
- a *navigation service* which provides the (non-SDK) APIs used by the two UIs, and can act as a backend for the various SDK geo-APIs; and
- a *routing engine* which calculates the recommendation for a route to a particular destination with particular parameters, and might be implemented in either the IVI or automotive domain. It is conceptually part of the *navigation service*.

These two UIs may be part of the same application, or may be separate applications (for example with the guidance UI as part of the system chrome). The navigation service may be a separate process, or may be combined with one or both of the UI processes.

The navigation service might communicate with systems in the automotive domain to provide its functionality.

Essentially, the 'navigation application'is a black box which provides UIs and (non-SDK) services related to navigation. For this reason, the rest of the document does not distinguish between 'navigation UI', 'guidance UI'and 'navigation service'.

Routing request

A *routing request* is a destination and set of optional parameters (waypoints, preferred options, etc.) from which the [routing engine][Navigation application] can calculate a specific route.

Use cases

A variety of use cases for application bundle usage of geo-features are given below. Particularly important discussion points are highlighted at the bottom of each use case.

In all of these use cases, unless otherwise specified, the functionality must work regardless of whether the vehicle has an internet connection. i.e. They must work offline. For most APIs, this is the responsibility of the automotive backend; SDK backend can assume an internet connection is always available.

Relocating the vehicle

If the driver is driving in an unfamiliar area and thinks they know where they are going, then realises they are lost, they must be able to turn on geolocation and it should pinpoint the vehicle's location on a map if it's possible to attain a GPS lock or get the vehicle's location through other means.

Automotive backend

A derivative of Apertis may wish to integrate its own geo-backend, running in the automotive domain, and providing all geo-functionality through proprietary interfaces. The system integrators may wish to use some functionality from this backend and other functionality from a different backend. They may wish to ignore some functionality from this backend (for example, if its implementation is too slow or is missing) and not expose that functionality to the app bundle APIs at all (if no other implementation is available).

Custom automotive backend functionality The proprietary geo-backend in a derivative of Apertis may expose functionality beyond what is described in this design, which the system integrator might want to use in their own application bundles.

If this functionality is found to be common between multiple variants, the official Apertis SDK APIs may be extended in future to cover it.

SDK backend

Developers using the Apertis SDK to develop applications must have access to geo-functionality during development. All geo-functionality must be implemented in the SDK.

The SDK can be assumed to have internet access, so these implementations may rely on the internet for their functionality.

Viewing an address on the map

The user receives an e-mail containing a postal address, and they want to view that address on a map. The e-mail client recognises the format of the address, and adds a map widget to show the location, which it needs to convert to latitude and longitude in order to pinpoint.

In order to see the surrounding area, the map should be a 2D top-down atlas-style view.

In order for the user to identify the map area in relation to their current journey, if they have a route set in the navigation application, it should be displayed as a layer in the map, including the destination and any waypoints. The vehicle's current position should be shown as another layer.

Adding custom widgets on the map

A restaurant application wants to display a map of all the restaurants in their chain, and wants to customise the appearance of the marker for each restaurant, including adding an introductory animation for the markers. They want to animate between the map and a widget showing further details for each restaurant, by flipping the map over to reveal the details widget.

Finding the address of a location on the map

The user is browsing a tourist map application and has found an interesting-looking place to visit with their friends, but they do not know its address. They want to call their friends and tell them where to meet up, and need to find out the address of the place they found on the map.

Type-ahead search and completion for addresses

A calendar application allows the user to create events, and each event has an address/location field. In order to ease entering of locations, the application wishes to provide completion options to the user as they type, if any potential completion addresses are known to the system's map provider. This allows the user to speed up entry of addresses, and reduce the frequency of typos on longer addresses while typing when the vehicle is moving.

If this functionality cannot be implemented, the system should provide a normal text entry box to the user.

Navigating to a location

A calendar application reminds the user of an event they are supposed to attend. The event has an address set on it, and the application allows the user to select that address and set it as the destination in their navigation application, to start a new navigation route.

Once the navigation application is started, it applies the user's normal preferences for navigation, and does not refer back to the calendar application unless the user explicitly switches applications.

Navigating a tour

A city tour guide application comes with a set of pre-planned driving tour routes around various cities. The driver chooses one, and it opens in the navigation application with the vehicle's current position, a series of waypoints to set the route, and a destination at the end of the tour.

At some of the waypoints, there is no specific attraction to see —merely a general area of the city which the tour should go through, but not necessarily using specific roads or visiting specific points. These waypoints are more like 'way-areas'.

Navigating to an entire city

The driver wants to navigate to a general area of the country, and refine their destination later or on-route. They want to set their destination as an entire city (for example, Paris; rather than 1 Rue de Verneuil, Paris) and have this information exposed to applications.

Changing destination during navigation

Part-way through navigating to one calendar appointment, the user gets a paging message from their workplace requiring them to divert to work immediately. The user is in an unfamiliar place, so needs to change the destination on their navigation route to take them to work —the paging application knows where they are being paged to, and has a button to set that as the new navigation destination. Clicking the button updates the navigation application to set the new destination and start routing there.

Navigating via waypoints The user has to stop off at their house on their way to work to answer the paging message. The paging application knows this, and includes the user's home address as a navigation waypoint on the way to their destination. This is reflected in the route chosen by the navigation application.

Tasks nearby

A to-do list application may allow the user to associate a location with a to-do list item, and should display a notification if the vehicle is driven near that location, reminding the driver that they should pop by and do the task. Once the task is completed or removed, the geo-fenced notification should be removed.

Turning on house lights

A 'smart home'application may be able to control the user's house lights over the internet. If the vehicle is heading towards the user's house, the app should be able to detect this and set turn the lights on over the internet to greet the user when they get home.

Temporary loss of GPS signal

When going through a tunnel, for example, the vehicle may lose sight of GPS satellites and no longer have a GPS fix. The system must continue to provide an estimated vehicle location to apps, with suitably increasing error bounds, if that is possible without reference to mapping data.

Navigation- and sensor-driven refinement of geolocation

The location reported by the geolocation APIs may be refined by input from the navigation system or sensor system, such as snapping the location to the nearest road, or supplementing it with dead-reckoning data based on the vehicle' s velocity history.

Application-driven refinement of geocoding

If the user installs an application bundle from a new restaurant chain ('Hamburger Co', who are new enough that their restaurants are not in commercial mapping datasets yet), and wants to search for such a restaurant in a particular place (London), they may enter 'Hamburger Co, London'. The application bundle should expose its restaurant locations as a general point of interest stream[2], and the geocoding system should query that in addition to its other sources.

The user might find the results from a particular application consistently irrelevant or uninteresting, so might want to disable querying that particular application —but still keep the application installed to use its other functionality.

Excessive results from application-driven refinement of geocoding

A badly written application bundle which exposes a general point of interest stream might return an excessive number of results for a query —either results which are not relevant to the current geographic area, or too many results to reasonably display on the current map.

Malware application bundle

A malicious developer might produce a malware application bundle which, when installed, tracks the user's vehicle to work out opportune times to steal it. This should not be possible.

Traffic rule application

The user is travelling across several countries in Europe, and finds it difficult to remember all the road signs, national speed limits and traffic rules in use in the countries. They have installed an application which reminds them of these whenever they cross a national border.

[2] https://www.apertis.org/concepts/archive/application/points_of_interest/#General_POI_providers

Installing a navigation application

The user wishes to try out a third-party navigation application from the Apertis store, which is different to the system-integrator-provided navigation application which came with their vehicle. They install the application, and want it to be used as the default handler for navigation requests from now on.

Third-party navigation-driven refinement of geolocation The third-party application has some advanced dead-reckoning technology for estimating the vehicle's position, which was what motivated the user to install it. The user wants this refinement to feed into the geolocation information available to all the applications which are installed.

Third-party navigation application backend If the user installs a full-featured third-party navigation application, they may want to use it to provide all geo-functionality in the system.

No navigation application

A driver prefers to use paper-based maps to navigate, and has purchased a non-premium vehicle which comes without a built-in navigation application bundle, and has not purchased any navigation bundles subsequently (i.e. the system has no navigation application bundle installed).

The rest of the system must still work, and any APIs which handle route lists should return an error code —but any APIs which handle the horizon should still include all other useful horizon data.

Web-based backend

An OEM may wish to use (for example) Google's web APIs for geo-services in their implementation of the system, rather than using services provided by a commercial navigation application. This introduces latency into a lot of the geo-service requests.

Navigation route guidance information

A restaurant application is running on one screen while the driver follows a route in their navigation application on another screen. The passenger is using the restaurant application to find and book a place to eat later on in the journey, and wants to see a map of all the restaurants nearby to the vehicle's planned route, plus the vehicle's current position, route estimates (such as time to the destination and time elapsed), and the vehicle's current position so they can work out the best restaurant to choose. (This is often known as route guidance or driver assistance information.)

While the passenger is choosing a restaurant, the driver decides to change their destination, or chooses an alternative route to avoid bad traffic; the passenger wants the restaurant application to update to show the new route.

2.5D or 3D map widget

A weather application would like to give a perspective view over a large area of the country which the vehicle's route will take it through, showing the predicted weather in that area for the next few hours. It would like to give more emphasis to the weather nearby rather than further away, hence the need for perspective (i.e. a 2.5D or 3D view).

Separate route guidance UI

An OEM wishes to split their navigation application in two parts: the navigation application core, which is used to find destinations and waypoints and to plan a route (including implementation of calculating the route, tracking progress through the journey, and recalculating in case of bad traffic, for example); and a guidance UI, which is always visible, and is potentially rendered as part of the system UI. The guidance UI needs to display the route, plus points of interest provided by other applications, such as restaurants nearby. It also needs to display status information about the vehicle, such as the amount of fuel left, the elapsed journey time, and route guidance.

Explicitly, the OEM does *not* want the navigation application core to display points of interest while the user is planning their journey, as that would be distracting.

User control over applications

The user has installed a variety of applications which expose data to the geo-services on the system, including points of interest and waypoint recommendations for routes. After a while, the user starts to find the behaviour of a fuel station application annoying, and while they want to continue to use it to find fuel stations, they do not want it to be able to add waypoints into their routes for fuel station stops.

Non-use-cases

The following use cases are not in scope to be handled by this design —but they may be in scope to be handled by other components of Apertis. Further details are given in each subsection below.

POI data

Use cases around handling of points of interest is covered by the Points of interest design[3], which is orthogonal to the geo-APIs described here. This includes searching for points of interest nearby, displaying points of interest while driving past them, adding points of interest into a navigation route, and looking up information about points of interest. It includes requests from the navigation application or guidance UI to the points of interest service, and the permissions system for the user to decide which points of interest should be allowed to appear in the navigation application (or in other applications).

Beacons

The iOS Location and Maps API supports advertising a device's location[4] using a low-power beacon, such as Bluetooth. This is not a design goal for Apertis at all, as advertising the location of a fast vehicle needs a different physical layer approach than Beacons, which are designed for low-speed devices carried by people.

Loading map tiles from a backend

There is no use case for implementing 2D map rendering via backends and (for example) loading map tiles *from a backend in the automotive domain*. 2D map rendering can be done entirely in the IVI domain using a single libchamplain tile source. At this time, the automotive domain will not carry 2D map tile data.

This may change in future iterations of this document to, for example, allow loading pre-rendered map tiles or satellite imagery from the automotive domain.

SDK APIs for third-party navigation applications

Implementing a navigation application is complex, and there are many approaches to it in terms of the algorithms used. In order to avoid implementing a lot of this complexity, and maintaining it as a stable API, the Apertis platform explicitly does not want to provide geo-APIs which are only useful for implementing third-party navigation applications.

Third parties may write navigation applications, but the majority of their implementation should be internal; Apertis will not provide SDK APIs for routing, for example.

[3] https://www.apertis.org/concepts/archive/application/points_of_interest/

[4] https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/RegionMonitoring/RegionMonitoring.html#//apple_ref/doc/uid/TP40009497-CH9-SW1

Requirements

Geolocation API

Geolocation using GPS must be supported. Uncertainty bounds must be provided for the returned location, including the time at which the location was measured (in order to support cached locations). The API must gracefully handle failure to geolocate the vehicle (for example, if no GPS satellites are available).

Locations must be provided with position in all three dimensions, speed and heading information if known. Locations should be provided with the other two angles of movement, the rate of change of angle of movement in all three dimensions, and uncertainty bounds and standard deviation for all measurements if known.

See Relocating the vehicle.

Geolocation service supports signals

Application bundles must be able to register to receive a signal whenever the vehicle's location changes significantly. The bundle should be able to specify a maximum time between updates and a maximum distance between updates, either of which may be zero. The bundle should also be able to specify a *minimum* time between updates, in order to prevent being overwhelmed by updates.

See Navigating to a location.

Geolocation implements caching

If an up-to-date location is not known, the geolocation API may return a cached location with an appropriate time of measurement.

See Relocating the vehicle, Tasks nearby.

Geolocation supports backends

The geolocation implementation must support multiple backend implementations, with the selection of backend or backends to be used in a particular distribution of Apertis being a runtime decision.

The default navigation application (requirement 5.6) must be able to feed geolocation information into the service as an additional backend.

See Automotive backend, Third-party navigation-driven refinement of geolocation

Navigation routing API

Launching the navigation application with zero or more waypoints and a destination must be supported. The navigation application can internally decide how to handle the new coordinates —whether to replace the current route, or supplement it. The application may query the user about this.

The interface for launching the navigation application must also support 'way-areas', or be extensible to support them in future. It must support setting some waypoints as to be announced, and some as to be used for routing but not as announced intermediate destinations.

It must support setting a destination (or any of the waypoints) as an address or as a city.

It must support systems where no navigation application is installed, returning an error code to the caller.

It must provide a way to request navigation routing suitable for walking or cycling, so that the driver knows how to reach a point of interest after they leave the vehicle.

See Navigating to a location, Changing destination during navigation, Navigating via waypoints, Navigating a tour, Navigating to an entire city, No navigation application.

Navigation routing supports different navigation applications

The mechanism for launching the navigation application (requirement 5.5) must allow a third-party navigation application to be set as the default, handling all requests.

See Installing a navigation application, Third-party navigation-driven refinement of geolocation.

Navigation route list API

A navigation route list API must be supported, which exposes information from the navigation application about the current route: the planned route, including destination, waypoints and way-areas.

The API must support signalling applications of changes in this information, for example when the destination is changed or a new route is calculated to avoid bad traffic.

If no navigation application is installed, the route list API must continue to be usable, and must return an error code to callers.

See Navigation route guidance information, No navigation application.

Navigation route guidance API

A route guidance API must be supported, which allows the navigation application to expose information about the directions to take next, and the vehicle's progress through the current route. It must include:

- Estimates such as time to destination and time elapsed in the journey. Equivalently, the journey departure and estimated arrival times at each destination.
- Turn-by-turn navigation instructions for the route.

The API must support data being produced by the navigation application and consumed by a single system-provided assistance or guidance UI, which is part of the system UI. It must support being called by the navigation application when the next turn-by-turn instruction needs to be presented, or the estimated journey end time changes.

See Navigation route guidance information.

Type-ahead search and address completion supports backends

The address completion implementation must support multiple backend implementations, with the selection of backend or backends to be used in a particular distribution of Apertis being a runtime decision.

See Automotive backend, Type-ahead search and completion for addresses.

Geocoding supports backends

The geocoding implementation must support multiple backend implementations, with the selection of backend or backends to be used in a particular distribution of Apertis being a runtime decision.

See Automotive backend.

SDK has default implementations for all backends

A free software, default implementation of all geo-functionality must be provided in the SDK, for use by developers. It may rely on an internet connection for its functionality.

The SDK implementation must support all functionality of the geo-APIs in order to allow app developers to test all functionality used by their applications.

See SDK backend.

SDK APIs do not vary with backend

App bundles must not have to be modified in order to switch backends: the choice of backend should not affect implementation of the APIs exposed in the

SDK to app bundles.

If a navigation application has been developed by a vendor to use vendor-specific proprietary APIs to communicate with the automotive domain, that must be possible; but other applications must not use these APIs.

See Automotive backend, Custom automotive backend functionality.

Third-party navigation applications can be used as backends

A third-party or OEM-provided navigation application must, if it implements the correct interfaces, be able to act as a backend for some or all geo-functionality.

See Third-party navigation application backend.

Backends operate asynchronously

As backends for geo-functionality may end up making inter-domain requests, or may query web services, the interfaces between applications, the SDK APIs, and the backends must all be asynchronous and tolerant of latency.

See the Inter-Domain Communications design.

See Web-based backend.

2D map rendering API

Map display has the following requirements:

- Rendering the map (see Relocating the vehicle).
- Rendering points of interest, including start and destination points for navigation.
- Rendering a path or route.
- Rendering a polygon or region highlight.
- The map display must support loading client side map tiles, or server-provided ones.
- The map rendering may be done client-side (vector maps) or pre-computed (raster maps).
- Rendering custom widgets provided by the application.
- Optionally rendering the current route list as a map layer.
- Optionally rendering the vehicle's current position as a map layer (see [Relocating the vehicle][Relocating the vehicle]).

See Viewing an address on the map, Adding custom widgets on the map.

2.5D or 3D map rendering API

For applications which wish to present a perspective view of a map, a 2.5D or 3D map widget should be provided with all the same features as the 2D map rendering API.

See 2.5D or 3D map widget.

Reverse geocoding API

Reverse geocoding must be supported, converting an address into zero or more coordinates. Limiting the search results to coordinates in a radius around a given reference coordinate pair must be supported.

Reverse geocoding may work when the vehicle has no internet connection, but only if that is easy to implement.

See Viewing an address on the map.

Forward geocoding API

Forward geocoding must be supported for querying addresses at selected coordinates on a map. Limiting the search results to a certain number of results should be supported.

Forward geocoding may work when the vehicle has no internet connection, but only if that is easy to implement.

See Finding the address of a location on the map.

Type-ahead search and address completion API

Suggesting and ranking potential completions to a partially entered address must be supported by the system, with latency suitable for use in a type-ahead completion system. This should be integrated into a widget for ease of use by application developers.

Address completion may work when the vehicle has no internet connection, but only if that is easy to implement.

This may need to be integrated with other keyboard usability systems, such as typing suggestions and keyboard history. If the functionality cannot be implemented or the service for it is not available, the system should provide a normal text entry box to the user.

See Type-ahead search and completion for addresses.

Geofencing API

Application bundles must be able to define arbitrary regions –either arbitrary polygons, or points with radii –and request a signal when entering, exiting, or dwelling in a region. The vehicle is dwelling in a region if it has been in there for a specified amount of time without exiting.

See Tasks nearby, Turning on house lights.

Geofencing service can wake up applications

It must be possible for geofencing signals to be delivered even if the application bundle which registered to receive them is not currently running.

See Tasks nearby, Turning on house lights.

Geofencing API signals on national borders

The geofencing API should provide a built-in geofence for the national borders of the current country, which applications may subscribe to signals about, and be woken up for as normal, if the vehicle crosses the country's border.

See Traffic rule application.

Geocoding API must be locale-aware

The geocoding API must support returning results or taking input, such as addresses, in a localised form. The localisation must be configurable so that, for example, the user's home locale could be used, or the locale of the country the vehicle is currently in.

See Traffic rule application.

Geolocation provides error bounds

The geolocation API must provide an error bound for each location measurement it returns, so calling code knows how accurate that data is likely to be.

See Temporary loss of GPS signal.

Geolocation implements dead-reckoning

The geolocation API must implement dead reckoning based on the vehicle's previous velocity, to allow a location to be returned even if GPS signal is lost. This must update the error bounds appropriately (Geolocation provides error bounds).

See Temporary loss of GPS signal.

Geolocation uses navigation and sensor data if available

If such data is available, the geolocation API may use navigation and sensor data to improve the accuracy of the location it reports, for example by snapping the

GPS location to the nearest road on the map using information provided by the navigation application.

See Navigation- and sensor-driven refinement of geolocation.

General points of interest streams are queryable

The general point of interest streams[5] exposed by applications must be queryable using a library API.

The approach of exposing points of interest via the geocoding system, as results for reverse geocoding requests, was considered and decided against. Reverse geocoding is designed to turn a location (latitude and longitude) into information describing the nearest address or place —not to a series of results describing every point of interest within a certain radius. Doing so introduces problems with defining the search radius, determining which of the results is the geocoding result, and eliminating duplicate points of interest.

See the Points of interest design[6].

Further requirements and designs specific to how applications expose such general points of interest streams are covered in the Points of Interest design.

See Application-driven refinement of geocoding.

Location information requires permissions to access

There are privacy concerns with allowing bundles access to location data. The system must be able to restrict access to any data which identifies the vehicle's current, past or planned location, unless the user has explicitly granted a bundle access to it. The system may differentiate access into coarse-grained and fine-grained, for example allowing application bundles to request access to location data at the resolution of a city block, or at the resolution of tens of centimetres. Note that fine-grained data access must be allowed for geofencing support, as that essentially allows bundles to evaluate the vehicle's current location against arbitrary location queries.

Application bundles asking for fine-grained location data must be subjected to closer review when submitted to the Apertis application store.

See Malware application bundle.

Open question: What review checks should be performed on application bundles which request permissions for location data?

[5] https://www.apertis.org/concepts/archive/application/points_of_interest/#General_POI_providers

[6] https://www.apertis.org/concepts/archive/application/points_of_interest/

Rate limiting of general point of interest streams

When handling general point of interest streams generated by applications, the system must prevent denial of service attacks from the applications by limiting the number of points of interest they can feed to the geolocation and other services, both in the rate at which they are transferred, and the number present in the system at any time.

See Excessive results from application-driven refinement of geocoding.

Application-provided data requires permissions to create

The user must be able to enable or disable each application from providing data to the system geo-services, such as route recommendations or points of interest, without needing to uninstall that application (i.e. so they can continue to use other functionality from the application).

See User control over applications.

Existing geo systems

This chapter describes the approaches taken by various existing systems for exposing sensor information to application bundles, because it might be useful input for Apertis'decision making. Where available, it also provides some details of the implementations of features that seem particularly interesting or relevant.

W3C Geolocation API

The W3C Geolocation API[7] is a JavaScript API for exposing the user's location to web apps. The API allows apps to query the current location, and to register for signals of position changes. Information about the age of location data (to allow for cached locations) is returned. Information is also provided about the location's accuracy, heading and speed.

Android platform location API

The Android platform location API[8] is a low-level API for performing geolocation based on GPS or visible Wi-Fi and cellular networks, and does not provide geofencing or geocoding features. It allows geolocation and cached geolocation queries, as well as signals of changes in location. Its design is highly biased towards making apps energy efficient so as to maintain mobile battery life.

Google Location Services API for Android

The Google Location Services API for Android[9] is a more fully featured API than the platform location API, supporting geocoding and geofencing in addi-

[7] http://www.w3.org/TR/geolocation-API/

[8] http://developer.android.com/guide/topics/location/strategies.html

[9] http://developer.android.com/training/location/index.html

tion to geolocation. It requires the device to be connected to the internet to access Google Play services. It hides the complexity of calculating and tracking the device's location much more than the platform location API.

It allows apps to specify upper and lower bounds on the frequency at which they want to receive location updates. The location service then calculates updates at the maximum of the frequencies requested by all apps, and emits signals at the minimum of this and the app's requested upper frequency bound.

It also defines the permissions required for accessing location data more stringently, allowing coarse- and fine-grained access.

iOS Location and Maps API

The iOS Location Services and Maps API is available on both iOS and OS X. It supports many features: geolocation, geofencing, forward and reverse geocoding, navigation routing, and local search.

For geolocation, it supports querying the location and location change signals, including signals to apps which are running in the background.

Its geofencing support is for points and radii, and supports entry and exit signals but not dwell signals. Instead, it supports hysteresis based on distance from the region boundary.

Geocoding uses a network service; both forward and reverse geocoding are supported.

The MapKit API[10] provides an embeddable map renderer and widget, including annotation and overlay support.

iOS (but not OS X) supports using arbitrary apps as routing providers for rendering turn-by-turn navigation instructions[11]. An app which supports this must declare which geographic regions it supports routing within (for example, a subway navigation app for New York would declare that region only), and must accept routing requests as a URI handler. The URIs specify the start and destination points of the navigation request.

It also supports navigation routing using a system provider, which requires a network connection. Calculated routes include metadata such as distance, expected travel time, localised advisory notices; and the set of steps for the navigation. It supports returning multiple route options for a given navigation.

The local search API[12] differs from the geocoding API in that it supports *types*

[10] https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/MapKit/MapKit.html#//apple_ref/doc/uid/TP40009497-CH3-SW1

[11] https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/ProvidingDirections/ProvidingDirections.html#//apple_ref/doc/uid/TP40009497-CH8-SW5

[12] https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/LocationAwarenessPG/EnablingSearch/EnablingSearch.html#//apple_ref/doc/uid/TP40009

of locations, such as 'coffee'or 'fuel'. As with geocoding, the local search API requires a network connection.

GNOME APIs

GNOME uses several libraries to provide different geo-features. It does not have a library for navigation routing.

GeoClue GeoClue[13] is a geolocation service which supports multiple input backends, such as GPS, cellular network location and Wi-Fi based geolocation.

Wi-Fi location uses the Mozilla Location Service[14] and requires network connectivity.

It supports geolocation signals[15] with a minimum distance between signals, but no time-based limiting. It does not support geofencing, but the developers are interested in implementing it.

GeoClue's security model allows permissions to be applied to individual apps, and location accuracy to be restricted on a per-app basis. However, this model is currently incomplete and does not query the system's trusted computing base (TCB) (see the Security design for definitions of the TCB and trust).

Geocode-glib Geocode-glib[16] is a library for forward and reverse geocoding. It uses the Nominatim API, and is currently hard-coded to query nominatim.gnome.org[17]. It requires network access to perform geocoding.

The Nominatim API[18] does not require an API key (though it does require a contact e-mail address), but it is highly recommended that anyone using it commercially runs their own Nominatim server.

geocode-glib is tied to a single Nominatim server, and does not support multiple backends.

libchamplain libchamplain[19] is a map rendering library, providing a map widget which supports annotations and overlays. It supports loading or rendering map tiles from multiple sources.

497-CH10-SW1

[13] http://freedesktop.org/wiki/Software/GeoClue/

[14] https://wiki.mozilla.org/CloudServices/Location

[15] http://www.freedesktop.org/software/geoclue/docs/gdbus-org.freedesktop.GeoClue2.Client.html#gdbus-signal-org-freedesktop-GeoClue2-Client.LocationUpdated

[16] https://gitlab.gnome.org/GNOME/geocode-glib

[17] https://bugzilla.gnome.org/show_bug.cgi?id=756311

[18] http://wiki.openstreetmap.org/wiki/Nominatim

[19] https://wiki.gnome.org/Projects/libchamplain

NavIt NavIt[20] is a 3D turn-by-turn navigation system designed for cars. It provides a GTK+ or SDL interface, audio output using espeak, GPS input using gpsd, and multiple map rendering backends. It seems to expose some of its functionality as a shared library (libnavit), but it is unclear to what extent it could be re-used as a component in an application, without restructuring work. It may be possible to package it, with modifications, as a third-party navigation application, or as the basis of one.

Navigation routing systems

Three alternative routing systems are described briefly below; a full analysis based on running many trial start and destination routing problems against them is yet to be done.

GraphHopper GraphHopper[21] is a routing system written in Java, which is available as a server or as a library for offline use. It uses OpenStreetMap data, and is licenced under Apache License 2.0.

OSRM OSRM[22] is a BSD-licensed C++ routing system, which can be used as a server or as a library for offline use. It uses OpenStreetMap data.

YOURS YOURS[23] is an online routing system which provides a web API for routing using OpenStreetMap data.

NavServer

NavServer is a proprietary middleware navigation solution, which accesses a core navigation system over D-Bus. It is designed to be used as a built-in navigation application.

It is currently unclear as to the extent which NavServer could be used to feed data in to the SDK APIs (such as geolocation refinements).

GENIVI

GENIVI implements its geo-features and navigation as various components under the umbrella of the IVI Navigation project[24]. The core APIs it provides are detailed below.

[20] https://www.navit-project.org/
[21] https://graphhopper.com/
[22] http://project-osrm.org/
[23] https://wiki.openstreetmap.org/wiki/YOURS
[24] https://at.projects.genivi.org/wiki/display/NAV/IVI+Navigation+Home

Navigation The navigation application is based on NavIt, using OpenStreetMap for its mapping data. It implements route calculation, turn-by-turn instructions, and map rendering.

It is implemented in two parts: a navigation service, which uses libnavit to expose routing APIs over D-Bus; and the navigation UI, which uses these APIs. The navigation service is implemented as a set of plugins for NavIt.

Fuel stop advisor The fuel stop advisor is a demo application which consumes information from the geolocation API and the vehicle API to get the vehicle's location and fuel levels, in order to predict when (and where) would be best to refuel.

POI service The points of interest (POI) service is implemented using multiple 'content access module'(CAM) plugins, each providing points of interest from a different provider or source. When searching for POIs, the service sends the query to all CAMs, with a series of attributes and values to match against them using a set of operators (equal, less than, greater than, etc.); plus a latitude and longitude to base the query around. CAMs return their results to the service, which then forwards them to the client which originally made the search request.

CAMs may also register categories of POIs which they can provide. These categories are arranged in a tree, so the user may limit their queries to certain categories.

Additionally, POI searches may be conducted against a given route list, finding points of interest along the route line, instead of around a single centre point.

Finally, it supports a 'proximity alert'feature, where the POI system will signal a client if the vehicle moves within a given distance of any matching POI.

Positioning The positioning API provides a large amount of positioning data: time, position in three dimensions, heading, rate of change of position in three dimensions, rate of change of angle in three dimensions, precision of position in three dimensions, and standard deviation of position in three dimensions and heading.

The service emits signals about position changes at arbitrary times, up to a maximum frequency of 10Hz. It exposes information about the number of GPS satellites currently visible, and signals when this changes.

Google web APIs

Google provides various APIs for geo-functionality. They are available to use subject to a billing scale which varies based on the number of requests made.

Google Maps Geocoding API The Google Maps geocoding API[25] is a HTTPS service which provides forward and reverse geocoding services, and provides results in JSON or XML format.

Forward geocoding supports address formats from multiple locales, and supports filtering results by county, country, administrative region or postcode. It also supports artificially boosting the importance of results within a certain bounds box or region, and returning results in multiple languages.

Reverse geocoding takes a latitude and longitude, and optionally a result type which allows the result to be limited to an address, a street, or country (for example).

The array of potential results returned by both forward and reverse geocoding requests include the location's latitude and longitude, its address as a formatted string and as components, details about the address (if it is a postal address) and the type of map feature identified at those coordinates.

The service is designed for geocoding complete queries, rather than partially-entered queries as part of a type-ahead completion system.

Google Places API The Google Places API[26] supports several different operations: returning a list of places which match a user-provided search string, returning details about a place, and auto-completing a user's search string based on places it possibly matches.

The search API supports returning a list of points of interest, and metadata about them, which are within a given radius of a given latitude and longitude.

The details API takes an opaque identifier for a place (which is understood by Google and by no other services) and returns metadata about that place.

The autocompletion API takes a partial search string and returns a list of potential completions, including the full place name as a string and as components, the portion which matched the input, and the type of place it is (such as a road, locality, or political area).

Google Maps Roads API The Google Maps Roads API[27] provides a snap to roads API, which takes a list of latitude and longitude points, and which returns the same list of points, but with each one snapped to the nearest road to form a likely route which a vehicle might have taken.

The service can optionally interpolate the result so that it contains more points to smoothly track the potential route, adding points where necessary to disambiguate between different options.

[25] https://developers.google.com/maps/documentation/geocoding/intro
[26] https://developers.google.com/places/web-service/
[27] https://developers.google.com/maps/documentation/roads/intro

Google Maps Geolocation API The Google Maps Geolocation API[28] provides a way for a mobile device (any device which can detect mobile phone towers or Wi-Fi access points) to look up its likely location based on which mobile phone towers and Wi-Fi access points it can currently see.

The API takes some details about the device's mobile phone network and carrier, plus a list of identifiers for nearby mobile phone towers and Wi-Fi access points, and the signal strength the device sees for each of them. It returns a best guess at the device's location, as a latitude, longitude and accuracy radius around that point (in metres).

If the service cannot work out a location for the device, it tries to geolocate based on the device's IP address; this will always return a result, but the accuracy will be very low. This option may be disabled, in which case the service will return an error on failure to work out the device's location.

Approach

Based on the Existing geo systems) and Requirements, we recommend the following approach for integrating geo-features into Apertis. The overall summary is to use existing freedesktop.org and GNOME components for all geo-features, adding features to them where necessary, and adding support for multiple backends to support implementations in the automotive domain or provided by a navigation application.

Backends

Each of the geo-APIs described in the following sections will support multiple backends. These backends must be choosable at runtime so that a newly installed navigation application can be used to provide functionality for a backend. Switching backends may require the vehicle to be restarted, so that the system can avoid the complexities of transferring state between the old backend and the new one, such as route information or GPS location history.

Applications must not be able to choose which backend is being used for a particular geo-function —that is set as a system preference, either chosen by the user or fixed by the system integrator.

If there are particular situations where it is felt that the application developer knows better than the system integrator about the backend to use, that signals a use case which has not been considered, and might be best handled by a revision of this design and potentially introducing a new SDK API to expose the backend functionality desired by the application developer.

Backends may be implemented in the IVI domain (for example, the default backends in the SDK must be implemented in the IVI domain, as the SDK has no other domains), or in the automotive domain. If a backend is implemented

[28]https://developers.google.com/maps/documentation/geolocation/intro

in the automotive domain, its functionality must be exposed as a proxy service in the IVI domain, which implements the SDK API. Communications between this proxy service and the backend in the automotive domain will be over the inter-domain communications link.

See the Inter-Domain Communications design

This IPC interface serves as a security boundary for the backend.

Third-party applications (such as navigation applications) may provide backends for geo-services as dynamically loaded libraries which are installed as part of their application bundle. As this allows arbitrary code to be run in the context of the geo-services (which form security boundaries for applications, see Systemic security), the code for these third-party backends must be audited and tested (Testing backends) carefully as part of the app store validation process.

Due to the potential for inter-domain communications, or for backends which access web services to provide functionality, the backend and SDK APIs must be asynchronous and tolerant of latency.

Backends may expose functionality which is not abstracted by the SDK APIs. This functionality may be used by applications directly, if they wish to be tied to that specific backend. As noted above, this may signal an area where the SDK API could be improved or expanded in future.

Navigation application

Throughout this design, the phrase *navigation application* should be taken to mean the navigation application bundle, including its UI, a potentially separate Route guidance ui and any agents or backends for Backends. While the navigation application bundle may provide backends which feed data to a lot of the geo-APIs in the SDK, it may additionally use a private connection and arbitrary protocol to communicate between its backends and its UI. Data being presented in the navigation application UI does not necessarily come from SDK APIs. See [this non-use-case][SDK APIs for third-party navigation applications].

A system might not have a navigation application installed (see Navigation routing API), in which case all APIs which depend on it must return suitable error codes.

2D map display

libchamplain[29] should be used for top-down map display. It supports map rendering, adding markers, points of interest (with explanatory labels), and start and destination points. Paths, routes, polygons and region highlights can be rendered as using Clutter API on custom map layers.

[29] https://wiki.gnome.org/Projects/libchamplain

libchamplain supports pre-rendered tiles from online (ChamplainNetworkTileSource) or offline (ChamplainFileTileSource) sources. It supports rendering tiles locally using libmemphis, if compiled with that support enabled.

On an integrated system, map data will be available offline on the vehicle's file system. On the Apertis SDK, an internet connection is always assumed to be available, so map tiles may be used from online sources. libchamplain supports both.

libchamplain supports rendering custom widgets provided by the application on top of the map layer.

Route list layer on the 2D map A new libchamplain layer should be provided by the Apertis SDK which renders the vehicle's current route list as an overlay on the map, updating it as necessary if the route is changed. If no route is set, the layer must display nothing (i.e. be transparent).

Applications can add this layer to an instance of libchamplain in order to easily get this functionality.

In order for this to work, the application must have permission to query the vehicle's route list.

Vehicle location layer on the 2D map A new libchamplain layer should be provided by the Apertis SDK which renders the vehicle's current location as a point on the map using a standard icon or indicator. The location should be updated as the vehicle moves. If the vehicle's location is not known, the layer must display nothing (i.e. be transparent).

Applications can add this layer to an instance of libchamplain in order to easily get this functionality.

In order for this to work, the application must have permission to query the vehicle's location.

2.5D or 3D map display

Our initial suggestion is to use NavIt[30] for 3D map display. However, we are currently unsure of the extent to which it can be used as a library, so we cannot yet recommend an API for 2.5D or 3D map display. Similarly, we are unsure of the extent to which route information and custom rendering can be input to the library to integrate it with other routing engines; or whether it always has to use routes calculated by other parts of NavIt.

Open question: Is it possible to use NavIt as a stand-alone 2.5D or 3D map widget?

[30]https://www.navit-project.org/

Geolocation

GeoClue[31] should be used for geolocation. It supports multiple backends, so closed source as well as open source backends can be used. Some of the advanced features which do not impact on the API could be implemented in an automotive backend, although other backends would benefit if they were implemented in the core of GeoClue instead. For example, cached locations and dead-reckoning of the location based on previous velocity for when GPS signal is lost.

> The vehicle's velocity may be queried from the sensors; see the Sensors and Actuators design

GeoClue supports geolocation using GPS (from a modem), 3G and Wi-Fi. It supports accuracy bounds for locations[32], but does not pair that with information about the time of measurement. That would need to be added as a new feature in the API. Speed, altitude and bearing information are supported[33]. The other two angles of movement, the rate of change of angle of movement in all three dimensions, and uncertainty bounds and standard deviation for non-position measurements are not currently included in the API, and should be added.

The API already supports signalling of failure to geolocate the vehicle, by setting its Location property to '/'(rather than a valid org.freedesktop.GeoClue2.Location object path).

If the navigation application implements a snap-to-road feature, it should be used as a further source of input to GeoClue for refining the location.

Geolocation signals GeoClue emits a LocationUpdated signal[34] whenever the vehicle's location changes more than the DistanceThreshold[35]. GeoClue currently does not support rate limiting emission of the LocationUpdated signal for minimum and maximum times between updates. That would need to be added to the core of GeoClue.

Navigation routing

Navigation routing will be implemented internally by the OEM-provided navigation application, or potentially by a third-party navigation application installed by the user. In either case, there will not be an Apertis SDK API for calculating routes.

[31] http://freedesktop.org/wiki/Software/GeoClue/

[32] http://www.freedesktop.org/software/geoclue/docs/gdbus-org.freedesktop.GeoClue2.Location.html#gdbus-property-org-freedesktop-GeoClue2-Location.Accuracy

[33] http://www.freedesktop.org/software/geoclue/docs/gdbus-org.freedesktop.GeoClue2.Location.html

[34] http://www.freedesktop.org/software/geoclue/docs/gdbus-org.freedesktop.GeoClue2.Client.html#gdbus-signal-org-freedesktop-GeoClue2-Client.LocationUpdated

[35] http://www.freedesktop.org/software/geoclue/docs/gdbus-org.freedesktop.GeoClue2.Client.html#gdbus-property-org-freedesktop-GeoClue2-Client.DistanceThreshold

However, the SDK will provide an API to launch the navigation application and instruct it to calculate a new route. This API is the content hand-over[36] API, where the navigation application can be launched using a nav URI. The nav URI scheme is a custom scheme (which must not be confused with the standard geo URI scheme[37], which is for identifying coordinates by latitude and longitude). See Appendix: nav URI scheme for a definition of the scheme, and examples.

When handling a content hand-over request, the navigation application should give the user the option of whether to replace their current route with the new route —but this behaviour is internal to the navigation application, and up to its developer and policy.

The behaviour of the navigation application if it is passed an invalid URI, or one which it cannot parse (for example, due to not understanding the address format it uses) is not defined by this specification.

As per the content hand-over design[38], the user may choose a third-party navigation application as the handler for nav URIs, in which case it will be launched as the default navigation application.

If no navigation application is installed, or none is set as the handler for nav URIs, the error must be handled as per the content hand-over design.

Navigation route list API

Separately from the content hand-over API for sending a destination to the navigation application (see Navigation routing), there should be a navigation route list API to expose information about the route geometry to SDK applications.

This API will be provided by the SDK, but implemented by one of many backends —the navigation application will be one such backend, but another backend will be available on the SDK to expose mock data for testing applications against. The mock data will be provided by an emulator; see Testing applications for more information.

These backends will feed into a system service which provides the SDK API to applications, and exposes:

- The planned route geometry, including destination, waypoints and way-areas.
- Potential alternative routes.

The API will not expose the vehicle's current position —that's done by the Geolocation. Similarly, it will not expose points of interest or other horizon data —that's done by the points of interest API[39]; or guidance information —that's done by the Navigation route guidance API).

[36] https://www.apertis.org/concepts/archive/application_framework/content_hand-over/
[37] http://tools.ietf.org/html/rfc5870
[38] https://www.apertis.org/concepts/archive/application_framework/content_hand-over/
[39] https://www.apertis.org/concepts/archive/application/points_of_interest/

The API should emit signals on changes of any of this information, using the standard org.freedesktop.DBus.Properties signals. We recommend the following API, exposed at the well-known name org.apertis.Navigation1:

```
/* This object should implement the org.freedesktop.DBus.ObjectManager standard
   API to expose all existing routes to clients.
*/
object /org/apertis/Navigation1/Routes {
  read-only i property CurrentRoute; /* index into routes, or negative for ‘no current route’ */
  read-only ao property Routes; /* paths of objects representing potential routes, with the most highly recommend
}

/* One of these objects is created for each potential route.
   Each object path is suffixed by a meaningless ID to ensure its uniqueness.
   Route objects are immutable once published. Any change should be done by
   adding a new route and removing the old one.
*/
object /org/apertis/Navigation1/Routes/$ID {
  read-only a(dd) property Geometry; /* array of latitude-longitude pairs in order, from the start point to the d
   read-only u property TotalDistance; /* in metres */
   read-only u property TotalTime; /* in seconds */
   read-only a{ss} property Title; /* mapping from language name to a human-
readable title of the route in this language. Language names are using the POSIX locale format with locale identi
  read-only a{ua{ss}} property GeometryDescriptions; /* array of pairs of index and description, which attach a d
}
```

> Syntax is a pseudo-IDL and types are as defined in the D-Bus specification, http://dbus.freedesktop.org/doc/dbus-specification.html#type-system

Navigation route list backends The backend for the route list service is provided by the navigation application bundle, which may be built-in or provided by a user-installed bundle. If no navigation bundle is installed, no backend is used, and the route list service must return an error when called.

On the Apertis SDK, a navigation bundle is not available, so a mock backend should be written which presents route lists provided by the developer. This will allow applications to be tested using route lists of the developer’s choice.

Navigation route guidance progress API

There should be a navigation route guidance API to expose information about progress on the current route.

This API will be provided as interfaces by the SDK, but implemented by the navigation application. The UI responsible for displaying progress information or third party applications can use this API to fetch the info from the navigation application.

The route guidance progress API should be a D-Bus API which exposes estimates such as time to destination and time elapsed in the journey.

The API should emit signals on changes of any of this information, using the standard `org.freedesktop.DBus.Properties` signals. We recommend the following API, exposed at the well-known name `org.apertis.NavigationGuidance1.Progress` on the object `/org/apertis/NavigationGuidance1/Progress`:

```
interface org.apertis.NavigationGuidance1.Progress {
 read-only t property StartTime; /* UNIX timestamp, to be interpreted in the local timezone */
 read-only t property EstimatedEndTime; /* UNIX timestamp, to be interpreted in the local timezone */
}
```

Additional properties may be added in future.

> Syntax is a pseudo-IDL and types are as defined in the D-Bus specification, http://dbus.freedesktop.org/doc/dbus-specification.html#type-system

Navigation route guidance turn-by-turn API

There should be a navigation route guidance API to allow turn-by-turn guidance notifications to be presented.

This API will be provided as interfaces by the SDK, but implemented by a system component which is responsible for presenting notifications —this will most likely be the compositor. The navigation application can then call the TurnByTurn API to display new turn-by-turn driving instructions.

The route guidance turn-by-turn API should be a D-Bus API which exposes a method for presenting the next turn-by-turn navigation instruction.

We recommend the following API, exposed at the well-known name `org.apertis.NavigationGuidance1.TurnByTurn` on the object `/org/apertis/NavigationGuidance1/TurnByTurn`:

```
interface org.apertis.NavigationGuidance1.TurnByTurn {
 /* See https://people.gnome.org/~mccann/docs/notification-spec/notification-
spec-latest.html#command-notify */
  method Notify (in u replaces_id,
                 in s icon_name,
                 in s summary,
                 in s body,
                 in a{sv} hints,
                 in i expire_timeout,
                 out u id)

/*  See  https://people.gnome.org/~mccann/docs/notification-spec/notification-
spec-latest.html\#command-close-notification */

  method CloseNotification (in u id)
```

}

> Syntax is a pseudo-IDL and types are as defined in the D-Bus specification, http://dbus.freedesktop.org/doc/dbus-specification.html#type-system

The design of the turn-by-turn API is based heavily on the freedesktop.org Notifications specification[40], and could share significant amounts of code with the implementation of normal (non-guidance-related) notifications.

Route guidance UI By using a combination of the navigation route guidance API and the points of interest API[41], it should be possible for an OEM to provide a route guidance UI which is separate from their main navigation application UI, but which provides sufficient information for route guidance and display of points of interest, as described in Separate route guidance UI.

There is no need for a separate API for this, and it is expected that if an OEM wishes to provide a route guidance UI, they can do so as a component in their navigation application bundle, or as part of the implementation of the system chrome (depending on their desired user experience). The only requirement is that only one component on the system implements the route guidance D-Bus interfaces described above.

Horizon API

The Horizon API is a shared library which applications are recommended to use when they want to present horizon data in their UI —a combination of the route list and upcoming points of interest. The library should query the Navigation route list API and points of interest APIs[42] and aggregate the results for display in the application. It is provided as a convenience and does not implement functionality which applications could not otherwise implement themselves. The library should be usable by applications and by system components.

Any OEM-preferred policy for aggregating points of interest may be implemented in the horizon library in order to be easily usable by all applications; but applications may choose to query the SDK route list and points of interest APIs directly to avoid this aggregation and implement their own instead.

To use the horizon API, an application must have permission to query both the route list API and the points of interest API.

What applications do with the horizon data once they have received it is up to the application —they may, for example, put it in a local cache and store it to prevent re-querying for historical data in future. This is entirely up to the application developer, and is out of the scope of this design.

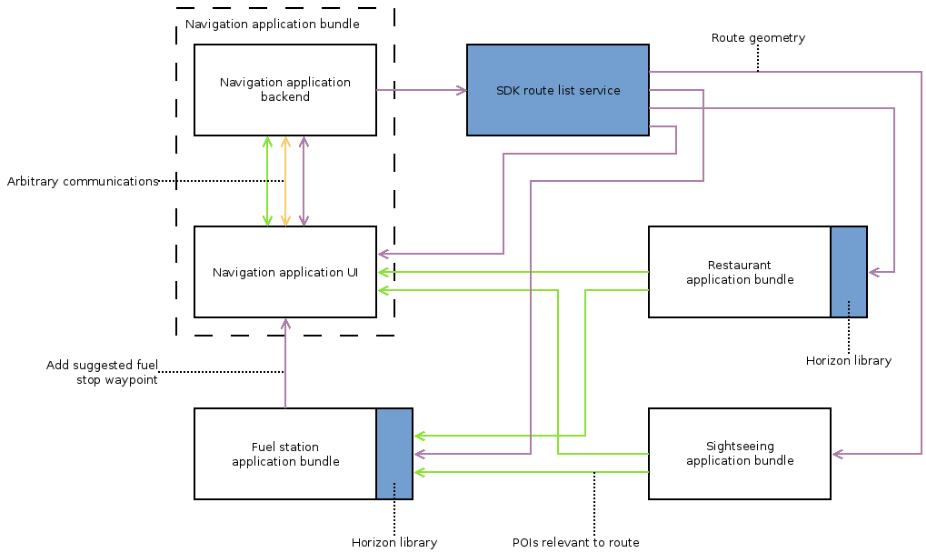
[40] https://specifications.freedesktop.org/notification-spec/notification-spec-latest.html
[41] https://www.apertis.org/concepts/archive/application/points_of_interest/
[42] https://www.apertis.org/concepts/archive/application/points_of_interest/

There was a choice between implementing the horizon API as a library or as a service. Implementing it as a service would not reduce memory consumption, as all consumers would likely still have to keep all horizon data in memory for rendering in their UI. It would not reduce CPU consumption, as aggregation of horizon data is likely to be simple (merging points of interest with the same name and location). It would double the number of IPC hops each piece of information had to make (changing from producer → consumer, to producer → horizon service → consumer). As all consumers of horizon data are likely to be interested in most points of interest, it would not significantly reduce the amount of data needing to be forwarded between producers and consumers. Hence a library is a more appropriate solution.

One potential downside of using a library is that it is harder to guarantee consistency in the horizon seen by different applications —the implementation should be careful to be deterministic so that users see the same horizon in all applications using the library.

See the following figure for a diagram of the flow of points of interest and route lists around the system. Key points illustrated are:

- Producers of points of interest choose which POIs are sent to which consumers (there is no central POI service), though the horizon library may perform filtering or aggregation according to system policy.
- The route list API is provided by the SDK, but uses one backend out of zero or more provided by the navigation application bundle or other bundles.
- The navigation UI is another application with no special powers; the arbitrary communications between its UI and backend may carry route lists, points of interest, or other data, but does not have to use the formats or APIs defined in the SDK.
- Applications choose which waypoints to send to via the [navigation routing API][Navigation routing] to be added into the route —this might result in the user being prompted 'do you want to add this waypoint into your route' , or they might be added unconditionally. For example, the fuel station application bundle may add a fuel stop waypoint to the route, which is then exposed in the route list API if the navigation application accepts it.
- The navigation UI does not necessarily use information from the route list API to build its UI (although such information may contribute).
- If no navigation bundle is installed, the SDK route list service still exists, it just returns no data.



Forward and reverse geocoding

We recommend that Geocode-glib[43] is used as the SDK geocoding API. Geocode-glib is currently hard-coded to use GNOME's Nominatim service; it would need to be modified to support multiple backends, such as an Apertis-specific Nominatim server[44], or a geocoding service from the automotive backend. It only needs to support querying one backend at a time; results do not need to be aggregated. Backends should be loadable and unloadable at runtime.

The geocode-glib API supports forward and reverse geocoding, and supports limiting search results to a given bounding box.

Geocoding backends On an integrated system, geocoding services will be provided by the automotive domain, via inter-domain communications (See the Inter-domain Communications design). On the Apertis SDK, an internet connection is always assumed to be available, so geocoding may be performed using an online Nominatim backend. In both cases, geocode-glib would form the SDK API used by applications. Another alternative backend, which could be written and used by OEMs instead of an automotive backend, would be a Google Maps geocoding API[45] backend which runs in the IVI domain.

Although it is tempting to do so, points of interest should *not* be fed into geocode-glib via another new backend, as the semantics of points of interest (a large number of points spread in a radius around a focal point) do not match the semantics of reverse geocoding (finding the single most relevant address to

[43] https://gitlab.gnome.org/GNOME/geocode-glib
[44] http://wiki.openstreetmap.org/wiki/Nominatim
[45] https://developers.google.com/maps/documentation/geocoding/intro

describe a given latitude and longitude). Points of interest should be queried separately using a library provided by the Points of Interest design[46].

Localisation of geocoding Nominatim supports exporting localised place names from OpenStreetMap, but geocode-glib does not currently expose that data in its query results. It would need to be modified to explicitly expose locale data about results.

It does currently support supplying the locale of input queries using the language parameter to geocode_forward_new_for_params[47].

Address completion

Address completion is a complex topic, both because it requires being able to parse partially-complete addresses, and because the datasets required to answer completion queries are large.

Nominatim does not provide dedicated address completion services, but it is possible to implement them in a separate web service using a filtered version of the OpenStreetMap database data. An example is available as Photon[48]. Google also provides a paid-for web service for address completion: the Google Places Web API[49].

As address completion is a special form of forward geocoding (i.e. forward geocoding operating on partial input), it should be provided as part of the geocoding service, and by the same backends which provide the geocoding functionality.

If Nominatim (via geocode-glib) is found to be insufficient for address completion in the SDK, an Apertis-hosted Photon instance could be set up, and a Photon backend added to the geocoding service.

In target devices, address completion should be provided by the automotive backend, or not provided at all if the backend does not implement it.

The address completion API should be an extension to the existing geocode-glib API for forward geocoding. There must be a way for it to signal there are no known results. Results should be ranked by relevance or likelihood of a match, and should include information about which part of the search term caused the match (if available), to allow that to be highlighted in the widget.

A separate library (which has a dependency on the SDK widget library) should provide a text entry widget which implements address completion using the API on the geocoding service, so that application developers do not have to reimplement it themselves. This could be similar to GtkSearchEntry[50] (but

[46] https://www.apertis.org/concepts/archive/application/points_of_interest/
[47] https://gitlab.gnome.org/GNOME/geocode-glib
[48] https://photon.komoot.io/
[49] https://developers.google.com/places/web-service/autocomplete
[50] https://developer.gnome.org/gtk3/stable/GtkSearchEntry.html

using the Apertis UI toolkit).

Address completion backends As the backends for the address completion service (i.e. the geocoding backends) may access sensitive data to answer queries, they must be able to check the permissions of the application which originated the query. If the application does not have appropriate permissions, they must not return sensitive results.

For example, a backend could be added which resolves 'home'to the user's home address, 'work'to their work address, and 'here'to the vehicle's current location. In order to maintain the confidentiality of this data, applications must have permission to access the system address book (containing the home and work addresses), and permission to access the vehicle's location (see Location security). If the application does not have appropriate permissions, the backend must not return results for those queries.

As normal geocoding operation is not sensitive (the results do not differ depending on who's submitting a query), backends which require permissions like this must be implemented in a separate security domain, i.e. as a separate process which communicates with geocode-glib via D-Bus. They can get the requesting application's unforgeable identifier from D-Bus requests in order to check permissions.

Geofencing

We recommend that GeoClue is modified upstream to implement geofencing of arbitrary regions, meaning that geofencing becomes part of Geolocation. Signals on entering or exiting a geofenced area should be emitted as a D-Bus signal which the application subscribes to. Delivery of signals to bundles which are not currently running may cause activation of that application.

> This is intended to be provided by a system service for activation of applications based on subscribed signals; the design is tracked in https://phabricator.apertis.org/T640.

The geofencing service should include a database of country borders, and provide a convenience API for adding a geofence for a particular country (or, for example, the current country and its neighbours). This would load the polygon for the country's border and add it as a geofence as normal.

The geofencing service must be available as the well-known name org.freedesktop.GeoClue2.Geofencing on D-Bus. It must export the following object:

```
/org/freedesktop/GeoClue2/Geofencing/Manager {
  /* Adds a geofence as a latitude, longitude and radius (in metres), and
   * returns the ID of that geofence. The dwell\_time (in seconds) gives
   * how long the vehicle must dwell inside the geofence to be signalled
   * as such by the GeofenceActivity signal. */
 method AddGeofenceByRadius(in (dd) location, in u radius, in u dwell_time, out u id)
```

```

  /* Adds a geofence by taking an ordered list of latitude and longitude
   * points which form a polygon for the genfence's boundary. */
  method AddGeofenceByPolygon(in a(dd) points, in u dwell\_time, out u id)

  /* Remove some geofences. */
  method RemoveGeofences(in au ids)

  /* Return a (potentially empty) list of the IDs of the geofences the
   * vehicle is currently dwelling in. */
  method GetDwellingGeofences(out au ids)

  /* Signal emitted every time a geofence is entered or exited, which
   * lists the IDs of all geofences entered and exited since the previous
   * signal emission, plus all the geofences the vehicle is currently
   * dwelling inside. */
  signal GeofenceActivity(out au entered, out au dwelling, out au exited)
}
```

IDs are global and opaque —applications cannot find the area referenced by a particular geofence ID. A geofence may only be removed by the application which added it. Currently, the GeofenceActivity signal is received by all applications, but they cannot dereference the opaque geofence identifiers for other applications. In future, if application-level containerisation is implemented, this signal will only be filtered per application.

We have informally discussed the possibility of adding geofencing with the GeoClue developers, and they are in favour of the idea.

Location security

libchamplain is a rendering library, and does not give access to sensitive information.

geocode-glib is a thin interface to a web service, and does not give access to sensitive information. All web service requests must be secured with best web security practices, such as correct use of HTTPS, and sending a minimum of identifiable information to the web service.

GeoClue provides access to sensitive information about the vehicle's location. It currently allows limiting the accuracy provided to applications as specified by the user[51]; this could be extended to implement a policy determined by the capabilities requested in the application's manifest.

Similarly for GeoClue's geofencing feature, when it is it added —clients have separated access to its D-Bus API to allow them to be signalled at different

[51] http://www.freedesktop.org/software/geoclue/docs/gdbus-org.freedesktop.GeoClue2.Client.html#gdbus-property-org-freedesktop-GeoClue2-Client.RequestedAccuracyLevel

accuracies and rates. This applies to navigation routing as well, as it may provide feedback to applications about progress along a route, which exposes information about the vehicle's location.

Application bundles asking for fine-grained location data should be subjected to closer review when submitted to the Apertis application store.

Systemic security

As the geo-features can source information from application bundles, they form part of the security boundary around application bundles.

In order to avoid denial of service attacks from an application bundle which emits too much data as, for example, a general point of interest stream, the system should rate limit such streams in time (number of POIs per unit time) and space (number of POIs per map area).

Location updates emitted by GeoClue must be rate limited between the minimum and maximum distance and time limits set by each client. These limits must be checked to ensure that a client is not requesting updates too frequently.

For the components in the system which provide access to sensitive information (the vehicle's location), a security boundary needs to be defined between them and application bundles. Geolocation, navigation route lists, route guidance and geofencing are the sensitive APIs —these are all implemented as services, so the D-Bus APIs for those services form the security boundary. In order to use any of these services, an application must have the appropriate permission in its manifest. Address completion as a whole is not treated as sensitive, but some of its backends may be sensitive, and perform their own checks according to other permissions (which may include those below). The following permissions are suggested:

- *content-hand-over*: Required to use the content hand-over API for setting a new [navigation route][Navigation routing].
- *location*: Required to access the geolocation, geofencing, or navigation route list services.
- *navigation-route*: Required to access the navigation route list services (note that the *location* permission is also required).
- *navigation-guidance*: Required to access the navigation guidance services (note that the *location* permission is also required).

The libchamplain layers which applications can use (see [here][Route list layer on the 2D map] and [here][Vehicle location layer on the 2D map]) are treated as application code which is using the relevant services (geolocation and navigation route guidance), and hence the *location* or *location* and *navigation-route* permissions are required to use them. Similarly for the horizon library.

Any service which accepts data from applications, such as points of interest or waypoints to add into the route list, must check that the user has not disabled that application from providing data. If the user has disabled it, the data must be ignored and an error code returned to the application if the API allows it, to indicate to the application that sharing was prevented.

Testability

There are several components to testability of geo-functionality. Each of the components of this system need to be testable as they are developed, and as new backends are developed for them (including testing the backends themselves). Separately, application developers need to be able to test their applications' behaviour in a variety of simulated geo-situations.

Testing geo-functionality Each of the services must have unit tests implemented. If the service has backends, a mock backend should be written which exposes the backend API over D-Bus, and integration tests for that service can then feed mock data in via D-Bus and check that the core of the service behaves correctly.

> Just like how libfolks'dummy backend is used in its unit tests, https://git.gnome.org/browse/folks/tree/backends/dummy

Testing backends Each service which has backends must implement a basic compliance test suite for its backends, which will load a specified backend and check that its public API behaves as expected. The default backends will further be tested as part of the integration tests for the entire operating system.

Testing applications In order to allow application developers to test their applications with mock data from the geo-services, there must be an emulator program available in the SDK which uses the mock backend for each service to feed mock data to the application for testing.

For example, the emulator program could display a map and allow the developer to select the vehicle's current location and the accuracy of that location. It would then feed this data to the mock backend of the geolocation service, which would pass it to the core of the geolocation service as if it were the vehicle's real location. This would then be passed to the application under test as the vehicle' s location, by the SDK geolocation API.

The emulator could additionally allow drawing a route on the map, which it would then send to the mock backend for the route list API as the current route —this would then be passed to the application under test as the vehicle's current route.

This means that the API of the mock backend for each service must be stable and well defined.

Requirements

This design fulfils the following requirements:

- Geolocation API —use GeoClue
- Geolocation service supports signals —use GeoClue; augment its signals
- Geolocation implements caching —to be added to GeoClue
- Geolocation supports backends —GeoClue supports backends
- Navigation routing API —use content hand-over design, passing a nav URI to the navigation application
- Navigation routing supports different navigation applications —content hand-over supports setting different applications as the handlers for the nav URI scheme
- Navigation route list API —new D-Bus API which is implemented by the navigation application backend
- Navigation route guidance API —new D-Bus API implemented by the system UI (i.e. the compositor) which is called by the navigation application
- Type-ahead search and address completion supports backends —implemented as part of geocoding, to be added to geocode-glib
- Geocoding supports backends —to be added to geocode-glib
- [SDK has default implementations for all backends][SDK has default implementations for all backends] —Gypsy or a mock backend for geolocation; custom online Nominatim server for geocoding; online OpenStreetMap for 2D maps; libnavit for 3D maps, **subject to further evaluation**; custom mock backend for navigation route list; custom online Nominatim or Photon server for address completion
- SDK APIs do not vary with backend —GeoClue API for geolocation; geocode-glib for geocoding; libchamplain for 2D maps; libnavit for 3D maps, **subject to further evaluation**; content hand-over API for navigation routing; new API for navigation route lists and guidance; new API extensions to geocode-glib for address completion
- Third-party navigation applications can be used as backends —backends are implemented as loadable libraries installed by the navigation application
- Backends operate asynchronously —backends are implemented over D-Bus so are inherently asynchronous
- 2D map rendering API —use libchamplain with a local or remote tile store
- 2.5D or 3D map rendering API —use libnavit, **subject to further evaluation**

- Reverse geocoding API —use geocode-glib
- Forward geocoding API —use geocode-glib
- Type-ahead search and address completion API —to be added to geocode-glib
- Geofencing API —to be implemented as a new feature in GeoClue
- Geofencing service can wake up applications —to be implemented as a new feature in GeoClue
- Geofencing API signals on national borders —to be added as a data set in GeoClue
- Geocoding API must be locale aware —to be added to geocode-glib to expose existing OpenStreetMap localised data
- Geolocation provides error bounds —GeoClue provides accuracy information, but it needs augmenting
- Geolocation implements dead reckoning —to be added to GeoClue
- Geolocation uses navigation and sensor data if available —to be added as another backend to GeoClue
- General points of interest streams are queryable —to be designed and implemented as part of the points of interest design[52]
- Location information requires permissions to access —to be implemented as manifest permissions for application bundles
- Rate limiting of general point of interest streams —security boundary implemented as D-Bus API boundary; rate limiting applied on signal emission and processed general point of interest streams
- Application provided data requires permissions to create —all geo-services must check settings before accepting data from applications

Suggested roadmap

As the SDK APIs for geo-features are, for the most part, provided by FOSS components which are available already, the initial deployment of geo-features requires GeoClue, geocode-glib and libchamplain to be packaged for the distribution, if they are not already.

The second phase would require modification of these packages to implement missing features and implement additional backends. This can happen once the initial packaging is complete, as the packages fulfil most of Apertis'requirements in their current state. This requires the address completion APIs to be added to to geocode-glib, and the geofencing APIs to be added to GeoClue, amongst other

[52]https://www.apertis.org/concepts/archive/application/points_of_interest/

changes. These API additions should be prioritised over other work, so that application development (and documentation about application development) can begin.

This second phase includes modifying the packages to be container-friendly, so that they can be used by compartmentalised apps without leaking sensitive data from one app to another. This requires further in-depth design work, but should require fairly self-contained changes.

The second phase also includes writing the new services, such as the Navigation route list API, the Navigation route guidance API and the Horizon API.

Open questions

1. What review checks should be performed on application bundles which request permissions for location data?
2. Is it possible to use NavIt as a stand-alone 2.5D or 3D map widget?

Summary of recommendations

As discussed in the above sections the recommendations are:

- Packaging and using libchamplain for 2D map display.
- Adding a route list layer for libchamplain.
- Adding a vehicle location layer for libchamplain.
- Packaging and using libnavit for 3D map display, **subject to further investigation**.
- Packaging and using GeoClue for geolocation. It needs measurement times, cached location support, dead-reckoning and more measurements and uncertainty bounds to be added upstream.
- Adding minimum and maximum update periods for clients to upstream GeoClue, alongside the existing distance threshold API for location update signals.
- Adding a new navigation application backend to GeoClue to implement snap-to-road refinement of its location.
- Adding a new mock backend to GeoClue for the SDK.
- Implementing a library for parsing place and nav URIs and formalising the specifications so they may be used by third-party navigation applications.
- Adding support for place and nav URIs to the content hand-over service (Didcot) and adding support for a default navigation application.
- Implementing a navigation route list service with support for loadable backends, including a mock backend for the SDK.

- Implementing a navigation route guidance API in the system compositor.
- Implementing a horizon library for aggregating and filtering points of interest with the route list, suitable for use by applications.
- Packaging and using geocode-glib for forward and reverse geocoding. It needs support for exposing localised place names to be added upstream.
- Adding support for loadable backends to geocode-glib, including a mock backend for the SDK.
- Auditing geocode-glib to ensure it maintains data privacy by, for example, using TLS for all requests and correctly checking certificates.
- Auditing GeoClue to ensure it maintains data privacy by, for example, using TLS for all requests and correctly checking certificates.
- Implementing an address completion API in geocode-glib and implementing it in the Nominatim and mock backends.
- Implementing an address completion widget based on the address completion API.
- Implementing a geofencing API in upstream GeoClue.
- Integrating the geo-services with the app service proxy to apply access control rules to whether applications can communicate with it to retrieve potentially sensitive location or navigation data. Only permit this if the appropriate permissions have been set on the application bundle's manifest.
- Implementing an emulator program in the SDK for controlling mock data sent by the SDK geo-APIs to applications under test.
- Providing integration tests for all geo-functionality.

Appendix: Recommendations for third-party navigation applications

While this design explicitly does not cover providing SDK APIs purely to be used in implementing navigation applications, various recommendations have been considered for what navigation applications probably should do. These are non-normative, provided as suggestions only.

- Support different types of vehicle —the system could be deployed in a car, or a motorbike, or a HGV, for example. Different roads are available for use by these different vehicles.
- Support calculating routes between the start, waypoints and destination using public transport, in addition to the road network. For example, this could be used to provide a comparison against the car route; or to

incorporate public transport schemes such as park-and-ride into route suggestions.

- Support audio turn-by-turn navigation, reading out instructions to the driver as they are approached. This allows the driver to avoid looking at the IVI screen to see the map, allowing them to focus on the road.
- Route guidance must continue even if another application takes the foreground focus on the IVI system, meaning the guidance system must be implemented as an agent.
- Support different optimisation strategies for route planning: minimal travel time, scenic route, minimal cost (for fuel), etc.
- In order to match the driver's view out of the windscreen, the map displayed when navigating should be a 3D projection to better emphasise navigational input which is coming up sooner (for example, the nearest junction or road signs).
- Support changing the destination mid-journey and calculating a new route.
- Support navigating via zero or more waypoints before reaching the final destination. Support adding new waypoints mid-journey.
- Provide the driver with up-to-date information about the estimated travel time and distance left on their route, plus the total elapsed travel time since starting the route. This allows the driver to work out when to take rest breaks from driving.
- Detect when the driver has taken a wrong turning while navigating, and recalculate the route to bring them back on-route to their destination, reoptimising the route for minimal travel time (or some other criterion) from their new location. The new route might be radically different from the old one if this is more optimal. The route recalculation should happen quickly (on the order of five seconds) so that the driver gets updated routing information quickly.
- Support recalculating and potentially changing a navigation route if traffic conditions change and make the original route take significantly longer than an alternative. There must be some form of hysteresis on these recalculations so that two routes which have very similar estimated travel times, but which keep alternating as the fastest, do not continually replace each other as the suggested route. The application may ask or inform the driver about route recalculations, as the driver may be able to assess and predict the traffic conditions better than the application.
- Provide information to the driver about the roads the vehicle is travelling along or heading towards and going to turn on to in future, such as the road name and number, and major towns or cities the road leads to. This allows

the driver to match up the turn-by-turn navigation directions with on-road signage. (This is often known as route guidance or driver assistance information.)

- If the driver takes a rest break during a navigation route, and turns the vehicle off, the application must give the driver the option to resume the navigation route when the vehicle is turned on again. The route must be recalculated from the vehicle's current location to ensure the resumed route is still optimal for current traffic conditions.
- Support cancelling the navigation when the vehicle is turned on again, at which point all navigation and turn-by-turn directions stop.
- If driven abroad, the navigation application should provide locale-sensitive navigation information, such as speed limits in the local units, and road descriptions which match the local signage conventions.
- Support feeding geolocation data in to the system geolocation service, if such data may be more precise than the raw GPS positions; for example, if it can be snapped to the nearest road.
- Support feeding other geo-information to the other geo-services, such as answering geocoding queries or performing geo-fencing. Support being a full replacement for the inbuilt navigation application and all the SDK services it provides.
- Query the system POI API for restaurants and toilets at times and frequencies suitable for recommending food or toilet breaks to the driver appropriately. Allow the driver to dismiss or disable these, or to change the intervals. Do not recommend a break if the journey is predicted to end soon. Launch the application which provided the POI if the user clicks on a POI in the navigation application, so that they can perform further actions on that POI (for example, if it's a restaurant, they could reserve a table). When POIs are displayed in the navigation application, they can be rendered as desired by the navigation application developer; when they are displayed in other applications, they are rendered as desired by those applications'developers.

Appendix: place URI scheme

The place URI scheme is a non-standard URI scheme suggested to be used within Apertis for identifying a particular place, including a variety of redundant forms of information about the place, rather than relying on a single piece of information such as a latitude and longitude. To refer to a location by latitude and longitude *only*, use the standard geo URI scheme[53].

A place URI is a non-empty human-readable string describing the location, followed by zero or more key–value pairs providing additional information. The

[53] http://tools.ietf.org/html/rfc5870

key–value pairs are provided as parameters as defined in [RFC 5870], i.e. each parameter is separated by a semicolon, keys and values are separated by an equals sign, and percent-encoding is used to encode reserved characters.

The location string must be of the format `1\*paramchar`, as defined in RFC 5870. All non-ASCII characters in the string must be percent-encoded[54], and implementations must interpret the decoded string as UTF-8[55].

Implementations may support the following parameters, and must ignore unrecognised parameters, as more may be added in future. All non-ASCII characters in parameter keys and values must be percent-encoded, and implementations must interpret the decoded strings as UTF-8. The semicolon and equals sign separators must not be percent-encoded. The ordering of parameters does not matter, unless otherwise specified. Each parameter may appear zero or one times, unless otherwise specified.

- `location`: the latitude and longitude of the place, as a geo URI (*with* the `geo:` scheme prefix)
- `postal-code`: the postal code for the place, in the country's postal code format
- `country`: the ISO 3166-1 alpha-2[56] country code
- `region`: the human-readable name of a large administrative region of the country, such as a state, province or county
- `locality`: the human-readable name of the locality, such as a town or city
- `area`: the human-readable name of an area smaller than the locality, such as a neighbourhood, suburb or village
- `street`: the human-readable street name for the place
- `building`: the human-readable name or number of a house or building within the `street`
- `formatted-address`: a human-readable formatted version of the entire address, intended to be displayed in the UI rather than machine-parsed; implementations may omit this if it is identical to the location string, but it will often be longer to represent the location unambiguously (the location string may be ambiguous or incomplete as it is typically user input)

Examples

This section is non-normative. Each example is given as a fully encoded string, followed by it split up into its un-encoded components.

- `place:Paris`

[54] http://tools.ietf.org/html/rfc5870#section-3.5
[55] http://www.unicode.org/versions/Unicode6.0.0/ch03.pdf
[56] http://www.iso.org/iso/country_codes.htm

    - Location string: Paris
    - No parameters

- `place:Paris;location=geo%3A48.8567%2C2.3508;country=FR;formatted-address=Paris%2C%20France`

    - Location string: Paris
    - Parameters:
        * `location`: `geo:48.8567,2.3508`
        * `country`: FR
        * `formatted-address`: Paris, France

- `place:K%C3%B6nigsstieg%20104%2C%2037081%20G%C3%B6ttingen;location=geo%3A51.540060%2C9.911850;country=`
  `code=37081;street=K%C3%B6nigsstieg;building=104;formatted-address=K%C3%B6nigsstieg%20104%2C%2037081%2`

    - Location string: Königsstieg 104, 37081 Göttingen
    - Parameters:
        * `location`: `geo:51.540060,9.911850`
        * `country`: DE
        * `locality`: Göttingen
        * `postal-code`: 37081
        * `street`: Königsstieg
        * `building`: 104
        * `formatted-address`: Königsstieg 104, 37081 Göttingen, Germany

- `place:CN Tower;location=geo%3A43.6426%2C-79.3871;formatted-address=301%20Front%20St%20W%2C%20Toronto%2`

    - Location string: CN Tower
    - Parameters:
        * `location`: `geo:43.6426,-79.3871`
        * `formatted-address`: 301 Front St W, Toronto, ON M5V 2T6, Canada

Appendix: nav URI scheme

The nav URI scheme is a non-standard URI scheme suggested to be used within Apertis for identifying a navigation route, including its destination, intermediate destinations (waypoints) and points or areas to route via but which are not named destinations (via-points). Each point or area may be provided as a place or a location.

A nav URI is a non-empty destination place, followed by zero or more key–value pairs providing additional information. The key–value pairs are provided as parameters as defined in [RFC 5870], i.e. each parameter is separated by a semicolon, keys and values are separated by an equals sign, and percent-encoding is used to encode reserved characters.

The destination place must be provided as Appendix: place URI scheme (*with* the `place:` URI prefix), or as a geo URI (*with* the `geo:` URI prefix); and must be encoded in the format `1\*paramchar`, as defined in RFC 5870; i.e. all non-ASCII and reserved characters in the string must be percent-encoded.

Implementations may support the following parameters, and must ignore unrecognised parameters, as more may be added in future. All non-ASCII characters in parameter keys and values must be percent-encoded, and implementations must interpret the decoded strings as UTF-8. The semicolon and equals sign separators must not be percent-encoded. The ordering of parameters does not matter, unless otherwise specified. Each parameter may appear zero or one times, unless otherwise specified.

- `description`: a human-readable description of the route, intended to be displayed in the UI rather than machine-parsed
- `way`: a named intermediate destination, as a place URI (*with* the `place:` scheme prefix) or as a geo URI (*with* the `geo:` scheme prefix); these parameters are order-dependent (see below)
- `via`: a non-named intermediate routing point, as a place URI (*with* the `place:` scheme prefix) or as a geo URI (*with* the `geo:` scheme prefix); these parameters are order-dependent (see below)

The `way` and `via` parameters are order-dependent: they will be added to the route in the order they appear in the nav URI. Way-places and via-places may be interleaved —they form a single route. The destination place always forms the final point in this route. The `way` and `via` parameters may each appear zero or more times.

Additional routing specific parameters can be added. If those parameters are not provided, the default value is left to the routing engine. It may be different for each type of vehicle, or due to other logic in the routing engine. Many parameters represent a single value; for example, it is not meaningful to specify both `optimize=fastest` and `optimize=shortest`. URIs with multiple values for a single-valued parameter, for example `place:Home;optimize=fastest;optimize=shortest`, should be treated as though that parameter was not present. Apertis does not currently define multi-valued preferences. All values should be specified at most once. However, OEMs may define and use their own multi-valued properties. The naming convention is defined below. Boolean values can be specified as `1` and `0` (or equivalently `true` and `false` while being case insensitive). Other values are considered invalid.

- `vehicle`: vehicle for which the route should be calculated. Apertis defines `car`, `walk`, and `bike`. `car` routes are optimized for being used by car. `walk` routes are optimized for walking. `bicycle` routes are optimized for being ridden by bicycles.
- `optimize`: Optimizes route calculation towards a set of criteria. Apertis defines `fastest`, and `shortest` criteria. `fastest` routes are optimized to minimize travel duration. `shortest` routes are optimized to minimize travel distance.

- `avoid-tolls`: Boolean. If true, the calculated route should avoid tolls. If usage can't be avoided, the handler application is responsible for informing the user.
- `avoid-motorways`: Boolean. If true, the generated route should avoid motorways. If usage can't be avoided, the handler application is responsible for informing the user.
- `avoid-ferries`: Boolean. If true, the generated route should avoid ferries. If usage can't be avoided, the handler application is responsible for informing the user.

Additionally, vendor specific parameters can be provided for vendor specific features. To avoid contradictory definitions for the same parameter in different implementations, vendor-specific parameters must be named in the form `x-vendorname-paramname`, similar to the convention for extending `.desktop` files[57]. Note that unlike keys in `.desktop` files, parameter names in `nav:` URIs are not case-sensitive: consistently using lower-case is encouraged. For instance, one of the [examples] below assumes that a manufacturer has introduced a boolean option `x-batmobile-avoid-detection`, and a string-valued parameter `x-batmobile-auto-pilot-mode`.

Examples

This section is non-normative. Each example is given as a fully encoded string, followed by it split up into its un-encoded components.

Example 1

`nav:place%3AKings%2520Cross%2520station%252C%2520London%3Blocality%3DLondon%3Bpostal-code%3DN19AL`

- Destination place:
  - Location string: Kings Cross station, London
  - Parameters:
    - `locality`: London
    - `postal-code`: N19AL

Example 2

`nav:geo%3A51.531621%2C-0.124372`

- Destination place: `geo:51.531621,-0.124372`

Example 3

`nav:place%3ABullpot%2520Farm%3Blocation%3Dgeo%253A54.227602%252C-2.517940;way=place%3ABirmingham%2520New%2520Street%2520station%3Blocation%3Dgeo%253A52.477620%252C-1.897904;via=place%3AHornby%3Blocation%3Dgeo%253A54.112245%252C-2.636527%253Bu%253D2000;way=place%3AInglesp`
`code%3DLA63EB%3Bcountry%3DGB`

- Destination place:

[57] https://specifications.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#extending

    - Location string: Bullpot Farm
    - Parameters:
        * location: geo:54.227602,-2.517940
- Parameters:
    - way:
        * Location string: Birmingham New Street station
        * Parameters:
            · location: geo:52.477620,-1.897904
    - via:
        * Location string: Hornby
        * Parameters:
            · location: geo:54.112245,-2.636527;u=2000
    - way:
        * Location string: Inglesport, Ingleton
        * Parameters:
            · street: The Square
            · building: 11
            · locality: Ingleton
            · postal-code: LA63EB
            · country: GB

Example 4 nav:geo%3A51.531621%2C-0.124372;vehicle=walk;optimize=shortest

- Destination place: geo:51.531621,-0.124372
- Parameters:
    - vehicle: walk
    - optimize: shortest

Example 5 nav:geo%3A51.531621%2C-0.124372;vehicle=car;avoid-tolls=false;x-batmobile-auto-pilot-mode=full;x-batmobile-avoid-detection=true

- Destination place: geo:51.531621,-0.124372
- Parameters:
    - vehicle: car
    - avoid-tolls: false
    - x-batmobile-avoid-detection: true
    - x-batmobile-auto-pilot-mode: full

Example 6 nav:place:Cambridge;x-myvendor-avoid-road=A14;x-myvendor-avoid-road=M11

- Destination place: Cambridge
- Parameters:
    - x-myvendor-avoid-road: multi-valued: A14, M11