Text To Speech

Contents

Text To Speech

Introduction

This documents possible approaches to designing an API for text to speech (TTS) services for an Apertis system in a vehicle.

This document proposes an API for the text to speech service in Appendix: A suggested TTS API. This API is not finalised, and is merely a suggestion. It may be refined in future versions of this document, or when implementation is started.

The major considerations with a TTS API are:

- Simple API for applications to use
- Swappable voices through the application bundling system and application store
- Output priorities controlled by the same set of audio manager policies which control other application audio output

Terminology and concepts

Text to speech (TTS)

Text to speech (TTS) is the process of converting a string of text into spoken words in the user's language, to be outputted as an audio stream.

Voice

In the context of TTS, a *voice* is an engine for producing spoken words. As with the conventional meaning of the word, the voice may have certain characteristics, such as gender, regionality or manners of speech. The most important quality of a voice is its understandability and correctness of pronunciation.

Use cases

A variety of use cases for application usage of TTS services are given below. Particularly important discussion points are highlighted at the bottom of each use case.

News application

The user has installed a news application, and wants it to read the headlines and articles aloud as they drive. If they are waiting in a traffic queue, they want to be able to quickly find the current paragraph in the article on-screen so they can read it themselves to speed things up.

Back in a news application

The user has a news reader application open on a specific article, which is being read aloud. The user presses the back button to close the article and return to the list of headlines. TTS output needs to stop for that article. If an audio source was playing before the user started reading the article (for example, some music), its playback may be resumed where it was paused.

New e-mail notification

The user's e-mail client is reading an e-mail aloud to the user, scrolling the e-mail as reading progresses. A new e-mail arrives, which causes a 'new e-mail' notification to be sent to the TTS system.

The OEM wants control over the policy of how the two TTS requests are played:

- The system could pause reading the original e-mail, read the notification, then resume reading the original e-mail; or
- it could pause reading the original e-mail, read the notification, then *not* resume reading the original e-mail; or
- it could continue reading the original e-mail at a lower volume, and read the notification louder mixed over the top.

The OEM wants these policies to not be overridable by any application-specific policy such as the ones described in New e-mail notification then going back, New meeting notification then cancelled, Incoming phone call.

New e-mail notification then going back

The user's e-mail client is reading an e-mail aloud to the user, scrolling the e-mail as reading progresses. A new e-mail arrives, which causes a 'new e-mail' notification to be sent to the TTS system. This pauses reading the original e-mail and starts reading the notification, as notifications have a higher priority than reading e-mails.

While the notification is being read, the user presses the 'back' button to go back to their list of e-mails. This should cancel reading out the old e-mail (which is currently paused), but should not cancel the 'new e-mail' notification, which is still being played.

New meeting notification then cancelled

The user's e-mail client is reading them an invitation to a meeting. While reading the invitation, the meeting is cancelled by the organiser, and a notification is displayed informing the user of this. This notification is read by the TTS system, interrupting it reading the original meeting invitation. Once the notification has finished being read, the e-mail client should not resume reading the original invitation.

Incoming phone call

The user's e-mail client is reading an e-mail aloud to the user. Part-way through reading, a phone call is received. TTS output for the e-mail needs to be automatically paused while the phone ringtone is played and the call takes place. Once the call has finished, the e-mail application may want to continue reading the user's e-mail aloud, or may cancel its output.

Voice installed with the SDK

A developer wants to develop an application using the SDK with TTS functionality, and needs to test it using a voice available in the SDK.

Installable voice bundle

A user does not like how the default TTS voice for their vehicle sounds, and wishes to change it to another voice which they can download from the Apertis application store. They wish this new voice to be used by default in future.

Voice backend in the automotive domain

An OEM may wish to provide a proprietary TTS voice as part of the software in their automotive domain. They want this voice to be used as the default for TTS requests from the CE domain as well.

Installable languages

A vehicle has already been released in various countries, but the OEM wishes to expand into other countries. They need to add support for additional languages to the TTS system.

Voice configuration

The user finds that the TTS system reads text too slowly for them, and they wish to speed it up. They edit their system preferences to increase the speed, and want this to take effect across all applications which use TTS.

Per-request emphasis

A news reader application needs to differentiate between TTS output for article headings and bodies. It wishes to read headings slightly louder and more slowly than it reads bodies. However, the application must not be allowed to make TTS requests so loud that they distract the driver.

Non-phonetic place names

The navigation application is reading turn-by-turn route guidance aloud, including place names. Various place names are not pronounced phonetically, and the navigation system needs to make sure the TTS system pronounces them correctly.

Driving abroad

When driving abroad, the navigation application needs to read the instructions "Turn left at the next junction, signposted 'Paris nord'.", a sentence which contains both English and French. The speech in each language should be pronounced using the correct pronunciation rules for that language.

Multiple concurrent TTS requests

The user is listening to their e-reader read a book aloud using TTS, while they are driving and using the audio turn-by-turn instructions from the navigation application. Whenever the navigation application needs to read an instruction, the e-reader output should be temporarily paused or its volume reduced, and resumed after the navigation instruction has been read, so that the user doesn't get confused.

It is understood that the current quality of TTS implementations is not sufficient to read an e-book to the user without causing them significant discomfort. This use case is intended to demonstrate the need for the system to handle multiple pending TTS requests. E-reader output may become possible in the future.

Permissions to use TTS API

The user has installed a game application for their passenger to play, and wants to be sure that it will not start reading instructions aloud using the TTS service while they are driving. They want to disallow the application permission to use the TTS API — either entirely, or just while driving.

Multiple output speakers

A vehicle has a single main field speaker, plus two sets of headphones. Each set of headphones is associated with a different head unit. TTS audio which pertains to the entire system should be output through all three speakers; TTS audio which pertains to an application only on one of the head units should only be output through that head unit's headphones.

Custom TTS implementation in an application

An application developer wants to port an existing application from another platform to Apertis. The application is a large one, and has its own tightly integrated TTS system which would output directly to the audio manager. This must be possible.

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.

Accessibility for users with reduced vision

While TTS is often used in software to provide accessibility for users with reduced vision, who otherwise cannot see the graphical UI clearly, that is not a goal of the TTS system in Apertis. It is intended to reduce driver distraction by reducing the need for the driver to look at the graphical UI, rather than making the UI more accessible.

Requirements

Basic TTS API

Implement a basic TTS API with support for speaking text; and pausing, resuming and cancelling specific requests.

See News application, Back in a news application, New e-mail notification then going back.

Progress signalling API

The TTS system must be able to signal an application as output progresses through the current request. Signals must be supported for output start and end, and may be supported for per-word progress through the text. Signals must also be supported for pausing and resuming output.

These signals are intended to be used to update the client application's UI to correspond to the output progress. For example, if a notification is being read aloud, the notification window should be hidden when, and only when, output is finished.

See News application, New e-mail notification then going back

Output policy decided by audio manager

The policy deciding which TTS requests are played, which are paused, when they are resumed, and which are cancelled altogether, must be determined by the system's audio manager.

An application may be able to implement its own policy (for example, to always cancel a TTS request if it is paused), but it must not be able to override the audio manager's policy, for example by preventing a request from being paused, or by increasing the priority of a request so it is played in preference to another.

If the audio manager corks a TTS output stream (for example, if all audio output needs to be stopped in order to handle a phone call), the TTS daemon must pause the corresponding client application request, and notify the application.

Once the output stream is uncorked, the client application request must be resumed, and the application notified, unless the application has cancelled that request in the meantime. By cancelling the request in the signal handler, a client application can ensure that TTS output is not resumed after the stream would have been uncorked, allowing for various resumption policies to be implemented.

See New e-mail notification then going back, New meeting notification then cancelled, Incoming phone call.

Output streams are mixable

Multiple TTS audio streams from within a single application, and from multiple applications, must be mixable by the audio manager, to allow implementing the policy of lowering the volume of one stream while playing a more important stream over the top.

See New e-mail notification.

Runtime-swappable voice backends

The TTS system must support different voice backends. Only one backend has to be active at once, but backends must be swappable at runtime if, for example, the user installs a new voice from the store, or if the OEM installs a voice backend supporting more languages (requirement 5.6).

TTS requests queued or being output at the time a new voice backend is selected should continue using the old voice. New TTS requests should use the new voice.

See Voice installed with the SDK, Voice configuration.

Installable voice backends

The user must be able to install additional voices from the Apertis application store; and an OEM must be able to install additional voices before sale of a vehicle to support additional languages. These voices must be available to choose as the default for all TTS output.

See Installable voice bundle, Installable languages.

Default SDK voice backend

A voice backend must be shipped with the SDK by default, to allow application development against the TTS system.

See Voice installed with the SDK.

Voice backends are not latency sensitive

Some vehicles may have a TTS voice backend implemented in the automotive domain, which means all TTS requests would be carried over the inter-domain communications link, incurring some latency. The TTS system must not be sensitive to this latency.

See Voice backend in the automotive domain.

System-wide voice configuration

The system must have a single default voice, which is used for all TTS output. The configuration settings for this voice must be settable in the system preferences, but not settable by individual applications.

Specific preferences, such as volume or speech rate, may be settable on a per-application basis to modify the system-wide defaults if needed. These modifications must have limited ability to distract the driver. For example, an application may apply a modifier to the volume of between 0.8 and 1.2 times the current system-wide output volume.

See Voice configuration.

Pronunciation data for non-phonetic words

There must be a way for applications to provide pronunciations for non-phonetic words. This may be implemented as a static list of overrides for certain words, or may be implemented as a runtime API. Pronunciations must be associated with a specific language, so that the correct pronunciation is used for the user's current system language. If no more suitable pronunciation is available for a word, the system must use the current voice's default pronunciation.

See Non-phonetic place names, Driving abroad.

Per-request language support

The TTS system must support specifying the language of each request (or even parts of a request), so that requests which contain text in multiple languages (for example 'Turn left onto Rue de Rivoli') are pronounced correctly.

The system language should be used by default if the application doesn't specify a language, or if the specified language is not supported by the current voice.

See Driving abroad.

Support for concurrent requests

The TTS system must support accepting TTS output requests from multiple applications concurrently, and queueing them for output sequentially.

See Multiple concurrent TTS requests.

Prioritisation for concurrent requests

The TTS system must support prioritising TTS requests from certain applications over requests from other applications, according to the urgency of the output (for example, turn-by-turn navigation instructions are more urgent than news reading). Similarly, it must support prioritising requests from within a single application.

Prioritisation must be performed on a per-request basis, as one application may make requests which are high and low priority. Note that this does not necessarily mean that the priority policy is implemented in the TTS system; it may be implemented in the audio manager. This requirement simply means that the TTS API must expose support for prioritising requests, and must forward that prioritisation information as 'hints' to whichever component implements the priority policy.

See Multiple concurrent TTS requests.

Output routing policy

On high-end vehicles, there may be multiple output speakers, attached to different head units. The audio manager must be able to associate each TTS request with an application so that it can determine which speaker or speakers to play the audio on.

See Multiple output speakers.

Permission for using TTS system

Applications must only be allowed to use the TTS system if they are allowed to output audio. This is subject to the application's permissions from its manifest, and may additionally be subject to the user's preferences for audio output. The user may be able to temporarily disable audio output for a specific application.

If any TTS-specific permissions are implemented in the system, it must be understood that an application may circumvent them by embedding its own TTS system (or by playing pre-recorded audio files, for example).

See Permissions to use TTS API, Custom TTS implementation in an application.

Existing text to speech systems

This chapter describes the approaches taken by various existing systems for allowing applications to use TTS services, 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.

Android

Android provides a text to speech API[1] for converting text to audio to output, or to audio in a file.

It provides an API for matching pieces of text with custom pre-recorded sounds (which it calls 'earcons'), for the purpose of embedding custom noises (such as ticking noises) into TTS output, or for providing custom pronunciations for the text.

It supports voices which support different languages, and provides the union of those languages to the developer, who may specify which language the provided text is in.

The user controls the preferences for the voice, apart from pitch and speech rate, which applications may set individually.

For determining the progress of the TTS engine through an utterance, the API provides a callback function which is called on starting and ending audio output.

[1] http://developer.android.com/reference/android/speech/tts/package-summary.html

iOS

iOS provides TTS support through its speech synthesiser API[2]. In this API, text to be spoken is passed to a new utterance object, which allows its voice, volume, speech rate and pitch to be modified. The utterance is then passed to the service, which queues it up to be spoken, or starts speaking it if nothing else is queued. Methods on the service allow output to be paused, cancelled or resumed. When pausing speech, the API provides the option to pause immediately, or after finishing speaking the current word.

Progress through speaking an utterance can be tracked using a delegate, which receives calls when speech starts, stops, is paused, resumes, and for each word in the text as it is spoken (intended for the purposes of highlighting words on-screen).

It is worth noting that iOS is recognised as highly competent in the field of accessibility for the blind or partially sighted, partly due to its well designed TTS system.

Previous eCore TTS API

The TTS API previously exposed by eCore gave a method to speak a given string of text, a method to stop speaking, and one to check whether speech was currently being output. It gave the choice of two voices, but no other options for configuring them. It provided two signals for notifying of audio output starting and ending.

speech-dispatcher

speech-dispatcher[3] is an abstraction layer over multiple TTS voices. It uses a client–server architecture, where multiple clients can connect and send text to the server to be outputted as audio. The protocol used between clients and the server is the Speech Synthesis Interface Protocol[4], a custom text-based protocol operated over a Unix domain socket.

Prioritisation between text from different clients is supported, but clients are not strictly separated by the server: one client can control the settings and output for another client.

The client library has C and Python APIs. The C API is pure C, and is not GLib-based. The backend supports a few different voices (see TTS voices): Festival, espeak, pico, and a few proprietary systems. Writing a new voice backend, to connect an existing external voice engine to speech-dispatcher, is not a major task.

[2] https://developer.apple.com/library/ios/documentation/AVFoundation/Reference/AVSpeechSynthesizer_Ref/index.html

[3] http://devel.freebsoft.org/speechd

[4] http://devel.freebsoft.org/doc/speechd/ssip.html

The system supports 'sound icons' which associate a sound file with a given text string, and allow that sound to be played when that string is found in input.

The settings allow control over the output language, whether to speak punctuation characters, the speech rate, pitch, and volume.

Speech output can be paused, resumed and cancelled once started. The API supports notifying when output is started, stopped, and when pre-defined 'index marks' are reached in the input string.

Backends for speech dispatcher are run as separate processes, communicating with the daemon via stdin and stdout. They have individual configuration files.

TTS voices

Here is a brief comparative evaluation of various TTS engines and voices which are available already.

espeak

- Supports many languages (importantly, non-Latin languages)
- Sounds robotic
- Can be used with mbrola voices to make it more natural; not supported very well by speech-dispatcher (http://espeak.sourceforge.net/mbrola.html)
- Already packaged for Ubuntu (as are mbrola voices)
- http://espeak.sourceforge.net/

Festival

- Sounds less robotic than espeak, but still quite robotic (example here[5])
- A bit slower
- Already packaged for Ubuntu
- Supports 3 languages (English, Spanish and Welsh)
- http://www.cstr.ed.ac.uk/projects/festival/

pico

- License: Apache License v2
- By SVOX; used in Android
- Written in Java; C API available in picoapi.h
- Supports 37 languages (importantly, non-Latin languages)
- Sounds very good (example here: https://svoxmobilevoices.wordpress.com/demos/)
- Not as well tested through speech-dispatcher
- https://en.wikipedia.org/wiki/SVOX

[5] https://www.cstr.ed.ac.uk/projects/festival/onlinedemo.html

- Publicly available source; https://android.googlesource.com/platform/external/svox/
- Already packaged for Debian and Ubuntu
- As this is a component of Android, we are not sure about the openness of the development practices, and whether it's possible to get involved in them.
- It's certainly possible to file bugs about the packaging with the Debian bug tracker[6], but that won't necessarily help for bugs in the source itself.

acapela

- Non-FOSS
- Best quality
- http://www.acapela-group.com/

Nuance

- Non-FOSS
- Has been used previously in eCore
- Demo[7]

Approach

Based on the above research (Existing text-to-speech systems) and Requirements, we recommend the following approach as an initial sketch of a text to speech system. A suggested API for the TTS service is given in Appendix: A suggested TTS API.

Overall architecture

As TTS output from an application is essentially another audio stream, and no privilege separation is required for turning a string of text into an audio stream, the design follows a 'decentralised' pattern similar to how GStreamer is implemented.

In order to produce TTS output, an application can link to a TTS library, which provides functionality for turning a text string into an audio stream. It then outputs this audio stream as it would any other, sending it to the audio manager, along with some metadata including an unforgeable identifier for the application, and potentially other metadata hints for debugging purposes. The audio manager applies the same priority policy which it applies to all audio streams, and determines whether to allow the stream to be played, pause it while another stream is played then resume it, or cancel it entirely. This is done using standard audio manager mechanisms using PulseAudio.

[6] https://bugs.debian.org/cgi-bin/pkgreport.cgi?pkg=libttspico0;dist=unstable

[7] https://www.nuance.com/omni-channel-customer-engagement/voice-and-ivr/text-to-speech.html#!

The TTS library receives feedback about the state of the audio channel, and passes this back to the application in the form of signals, which the application may use to update its UI, or implement its own policy for enqueuing or cancelling requests (or it may ignore the signals).

Alternative centralised design

The other major option is for a centralised design, where all TTS requests are sent to a TTS service (running as a separate process), which decides on relative priorities for them, converts them from text to audio, and forwards them to the audio manager.

There is no need for this design: there is no need for the additional privilege separation, and it complicates the application of audio policy, since it now has to be applied in the TTS service *and* the audio manager.

Use of speech-dispatcher

Speech dispatcher is an existing FOSS system which is the standard choice for systems like this. However, it is based around a centralised design which does not fit with our suggested architecture — a large part of speech-dispatcher is concerned with implementing a central daemon which handles connections and requests from multiple clients, prioritises them, then outputs them to the audio manager. As described in Overall architecture and Alternative centralised design, this is functionality which our recommended design does not need.

Additionally, speech-dispatcher has the disadvantages that it:

- does not enforce separation between clients, meaning they may control each others' output; and
- provides a C API which is not GLib-based, so would be hard to introspect and expose in other languages (such as JavaScript).

For these reasons, and due to its centralised architecture, we recommend *not* using speech-dispatcher. However, it may be possible and useful to extract relevant parts of its code and turn them into shared libraries to be used in the Apertis TTS library. The rest of this document will cover the design with no reference to speech-dispatcher, in the knowledge that it might substitute for some of the implementation work where possible.

TTS library

The TTS library would be a new shared library which can be linked into applications to essentially provide the functionality of turning a text string into an audio stream. It would provide the following major APIs:

- Say a text string.
- Stop, pause and resume speech.

- Signal on starting, pausing, resuming and ending audio output, plus on significant progress through output.
- Set the language for a request.
- 'Sound icon' API for associating audio files with specific strings.

The stop, pause and resume APIs would operate on specific requests, rather than all pending requests from the application. This allows for an application to cancel one TTS output while continuing to say another; or to cancel one output while another is paused. The API should be implemented as a queue-based one, where the application enqueues a string to be read, and receives a handle identifying it in the queue. The TTS library can prioritise requests within the queue, and hence requests may not be processed for some time after being enqueued. Signals convey this information to the application.

The progress signal should be emitted at the discretion of the TTS library, to signal significant progress to the application in outputting the TTS request. For example, it could be emitted once per sentence, or once per word, or not at all. It returns an offset (in Unicode characters) from the start of the input text.

The library's audio output would provided in a format suitable for passing directly to PulseAudio, or into GStreamer for further processing.

The TTS library would implement loading of a TTS backend into the process, and would load and apply the system settings for TTS output.

Installable and swappable backends

The TTS library would implement voice backends as dynamically loaded shared libraries, all installed into a common directory. It must monitor this directory at runtime to detect newly installed voice backends; for an application bundle to install a new backend, it would have to install or symlink the library into this directory.

The TTS library should not care how a voice backend is implemented internally, as long as it implements a standard backend interface. It may be possible, for example, to re-use a lot of the code from speech-dispatcher's backend modules[8].

Each voice backend must provide an interface for converting text to audio, and returning that audio to the TTS library — it should *not* implement outputting the audio to the audio manager itself. Backends must provide a way of enumerating and configuring their voice options (such as volume, pitch, accent, etc.), including a way of specifying that an option is read-only or unsupported. It is not expected that all backends will support all functionality of the TTS library.

The backend interface must be tolerant of latency in the backends, in order to support backends which are implemented in the automotive domain. This means that all functions must be asynchronous[9].

[8] http://git.freebsoft.org/?p=speechd.git;a=tree;f=src/modules;hb=HEAD
[9] https://developer.gnome.org/gio/stable/GAsyncResult.html

SDK default backend

We recommend Pico[10] as the default backend to ship with the SDK. It is freely licenced, and supports 37 languages including non-Latin languages. It is used on Android, so is relatively stable and mature.

Global configuration

Configuration options for the voice backends should be stored in GSettings (See Preferences and Persistence[11]), and should be stored once (not per-backend). The semantics of each configuration option must be rigorously defined, as each backend must convert those options to fit its own configuration interface. If a backend has more options in its configuration interface than are provided by the global TTS library configuration, it must use sensible, unconfigurable, defaults for the other options.

Configuration options may include:

- Voice to use
- Whether to vocalise punctuation
- Voice type (male or female)
- Speech rate
- Pitch
- Volume

By storing the options in GSettings, it becomes possible to apply AppArmor policy to control access to them so that, for example, applications which use the TTS library are only allowed to read the settings, and only the system preferences application is allowed to modify them.

Per-request configuration

Configuration which is exposed to applications via the TTS API could be:

- Pitch
- Speech rate
- Volume

These options must be exposed purely as *modifiers* on the system-wide values. These modifiers could be defined symbolically, for example as a set of three volume modifiers:

- Emphasised (120% of system-wide volume)
- Normal (100% of system-wide volume)
- De-emphasised (80% of system-wide volume)

A non-symbolic numerical modifier might be introduced in future.

[10] https://android.googlesource.com/platform/external/svox/
[11] https://martyn.pages.apertis.org/apertis-website/concepts/preferences-and-persistence/

The audio manager is responsible for limiting the maximum volume of any audio stream, to avoid a malicious or faulty application from setting the volume too high as to distract the driver.

Sound icons

Sound icons are a feature provided by speech-dispatcher, which we could use as the basis for our own implementation, as this would allow re-use of the relevant features in voice backends.

Sound icons could be used for identifying punctuation, for example, or for clarifying the pronunciation of certain words. It's suggested that applications install sound icons at install time, in a per-application directory which the application points the TTS library at to look up when asked to play a sound icon. Each sound icon should have an associated language (or explicitly no associated language), so that the correct sound icon file can be loaded according to a TTS request's language.

Sound icons should be playable via a TTS library API, similarly to how text output is requested. They should be provided in WAV format, as this is what the existing speech-dispatcher backends expect.

Request prioritisation

There are two dimensions to prioritisation of requests: within a single application, and across multiple applications.

Requests from within a single application should be handled using a request queue within the TTS library. This allows squashing similar requests, or bumping other requests so they are played before other requests from the same application.

It is suggested that the speech-dispatcher priorities[12] are used for within a single application, including their semantics. For example, the TTS library request queue would squash multiple progress requests so that only one is played at once.

These priorities should be attached to audio output when it is sent to the audio manager, as a hint to assist it in its policy decisions.

Requests from multiple applications are prioritised by the audio manager, which uses the audio priority of each application (whether it is an entertainment or interrupt source, and its numerical audio priority) from the application's manifest to determine which requests to play, which to pause then resume, and which to cancel entirely. The application's audio priority is under the control of the OEM, rather than the application developer, so application developers cannot use this to always output audio at an inflated priority and deny other applications audio output.

[12] http://devel.freebsoft.org/doc/speechd/ssip.html#Priority-Categories

See the Audio Management design

There is one situation where an application with a low priority may need to output a TTS request at a higher overall priority than an application with a high priority: when emitting a pop-up notification via the notification service. This should be handled by having notifications submitted as TTS requests by the notification service itself, rather than by the application which produced the notification. This allows the audio manager to use the notification service's priority for policy decisions, rather than the original application's priority.

PulseAudio output

Output from the TTS library should be sent to PulseAudio in order to be mixed with other TTS and non-TTS audio streams and sent to the hardware for output. It is PulseAudio and the audio manager which implement the priority policies described above.

In order to differentiate TTS output from different applications, appropriate metadata should be attached to the audio stream to identify the application, its internal priority for the TTS request, and the fact that the audio is a TTS request (as opposed to other audio content). The application identifier must be unforgeable (i.e. it must come from a trusted source, like the kernel or D-Bus daemon), as it is used as the basis for policy decisions. The internal priority and TTS request flag are entirely under the control of the application (i.e. forgeable), and therefore must only be used as hints by the audio manager. Additional unforgeable metadata may come from the application's manifest file, which is not under the control of the application developer, and can be uniquely looked up by the application's trusted identifier.

The audio manager most likely will *not* use forgeable metadata from the application, but this data could be useful for identifying audio streams when debugging, for example.

If an application wishes to submit multiple TTS requests simultaneously, and have the audio manager mix them or decide which one to prioritise, it must have multiple connections to PulseAudio.

If, as a result of applying the priority policy, the audio manager corks an application's TTS output stream, the TTS library must pause the corresponding TTS request and notify the application using a signal. Once the request is uncorked, the TTS library must unpause the request and notify the application again — unless the application has cancelled the request in the meantime, in which case the request is already cancelled and removed.

The same is true if the audio manager *cancels* an application's TTS output stream: the TTS library must cancel the corresponding TTS request and notify the application using a signal.

Note that the audio manager's pausing and resuming of TTS requests is separate

from the pause and resume APIs available to the application. The application cannot call its resume method to resume a TTS request which the audio manager has paused. Similarly, the audio manager cannot call its resume method to resume a TTS request which the application has paused. This can be thought of as separately pausing or resuming both ends of the audio channel between an application and the audio manager.

Testability

Testing the TTS system can be split into three major areas: checking that the TTS library and its various voice backends work; checking that the audio manager correctly applies its priority policies to incoming TTS audio streams and normal audio streams; and integration testing of audio output from an application calling a TTS API.

The former can be achieved using unit tests within the TTS library project, which test various components of the library in isolation. For example, they could compare TTS audio output streams against stored 'golden' expected output sound files.

The audio manager testing should be implemented as part of the audio manager's test plan, ensuring that TTS audio channel metadata is included in a variety of test situations.

This should be described in the Audio Management design.

Finally, the integration testing requires the audio output to be checked, so is infeasible to implement as an automated test, and would have to be a manual test where the human tester verifies that the output sounds as expected for a given set of input situations (requests from a test client).

Security

The security properties being implemented by the system are:

- Applications should be independent, in that one application cannot change the TTS settings for another application, or affect another application's TTS output other than through prioritisation of requests as controlled by the audio manager.
- Applications must not be able to play a TTS request if the audio manager has disallowed or paused it (availability of audio output to other applications).
- Applications should not be able to set the system-wide TTS preferences.
- Applications should not be able to determine the content of other applications' TTS requests (confidentiality of requests).
- Applications must only be allowed to use the TTS system if they have permission to output audio.

These are implemented through the separation of audio priority policies from the TTS library, by implementing them in the audio manager. The audio manager has a non-forgeable identifier for the application which originated each TTS audio stream, and the forgeable priority hints which come from the application are not allowed to override the application's audio priority.

Audio output from an application is subject to that application having permission to output audio, which is enforced by the audio manager.

Independence and confidentiality of application audio channels is implemented as for all audio channels, by having separate connections from each application to the audio manager.

Integrity of system-wide TTS preferences is implemented by the AppArmor policy controlling access to those preferences in GSettings.

Loadable voice backends

The TTS library, and hence each application which links to it, needs read-only and execute access to the loadable voice backend libraries, plus any resources needed by those voices. It also needs read-only access to the TTS system-wide preferences in GSettings.

Suggested roadmap

There are few opportunities for splitting this system up into stages. The TTS library needs to be written first, including its loadable voice backend interface and the first voice backend. More complex features like sound icons could be ignored in the first version of the library. With this working, applications could start to use the TTS APIs. The unit tests and integration tests for the TTS library should be written from the very beginning.

With TTS output working, a second stage could implement the priority policies in the audio manager, and ensure those are working. The system preferences could also be integrated at this stage.

A third stage could produce more voice backends (if needed), potentially including a voice backend which is implemented in the automotive domain, to ensure that asynchronous calls to the backends work.

It is worth highlighting that aside from initially ignoring features like sound icons, there is little scope for simplifying the TTS API for its first implementation. Specifically, we feel it would be a mistake to implement a non-queue-based API for scheduling TTS requests to begin with, and then 'expand' it into a queue-based API later on. To do so would expose applications to a lot of semantic changes in the API which they would then have to adapt to use. The TTS library API should be implemented as a queue-based one from the start.

Requirements

- Basic TTS API: Implemented as a C API on the TTS library.
- Progress signalling API: Implemented using GObject signals emitted by the TTS library.
- Output policy decided by audio manager: Implemented by passing priority and application identifiers to the audio manager, and it corking, uncorking, or cancelling audio streams according to its policy, using standard PulseAudio functionality.
- Output streams are mixable: Audio manager may choose to *not* cork two streams, and mix them instead.
- Runtime-swappable voice backends: TTS library loads backends from a directory as dynamically loaded libraries, and monitors that directory for changes.
- Installable voice backends: Installed or symlinked into the backend library directory.
- Default SDK voice backend: Pico to be shipped as the default backend for the SDK.
- Voice backends are not latency sensitive: Voice backend interface uses asynchronous functions to avoid blocking the TTS library.
- System-wide voice configuration: Stored in GSettings and read by the TTS library in each application which uses it. The system preferences application can modify the settings in GSettings.
- Pronunciation data for non-phonetic words: Provided by an API in the TTS library similar to the speech-dispatcher API for 'sound icons'.
- Per-request language support: Provided as a per-request API to hint at the language the source text is written in.
- Support for concurrent requests: Implemented by allowing multiple audio channel connections to the audio manager, which prioritises between them.
- Prioritisation for concurrent requests: Implemented by allowing multiple audio channel connections to the audio manager, which prioritises between them. In-application priorities are handled by a per-application request queue within the TTS library.
- Permission for using TTS system: Checked by the audio manager for each application which attempts to play audio (including TTS output), using permissions from the application's manifest.

Summary of recommendations

As discussed in the above sections, we recommend:

- Implementing a new TTS library, using an API like the one suggested in Appendix: A suggested TTS API. Parts of speech-dispatcher may be used to aid the implementation if appropriate.
- Implementing voice backends as dynamically loaded libraries, potentially reusing much of the existing backends from speech-dispatcher.
- Modifying the audio manager to support applying a priority policy to TTS requests, using the application's audio priority, and potentially logging TTS-specific metadata for debugging purposes.
- Implementing unit and integration tests for the TTS library, audio manager and TTS system as a whole.
- Packaging and using Pico as the default voice backend in the SDK.
- Modifying the Apertis software installer to generate AppArmor rules to allow access to the TTS voice backends and their resources, plus the TTS system settings, if an application is allowed to output audio.

Appendix A: Suggested TTS API

The code listing is given in pseudo-code.

```
/* TTS context to contain relevant state and loaded resources and
 * settings. */
class TtsContext {
  async TtsRequest send_request (const string text_to_say,
                                 TtsPriority priority=TEXT,
                                 const string language=null,
                                 TtsVoiceRate voice_rate=TtsVoiceRate.NORMAL,
                                 TtsVolume volume=TtsVolume.NORMAL,
                                 TtsPitch pitch=TtsPitch.NORMAL);

  async TtsRequest send_sound_icon_request (const string icon_name,
                                            TtsPriority priority=TEXT,
                                            const string language=null,
                                 TtsVoiceRate voice_rate=TtsVoiceRate.NORMAL,
                                           TtsVolume volume=TtsVolume.NORMAL,
                                            TtsPitch pitch=TtsPitch.NORMAL);
}

/* This represents a single pending TTS request. The object may persist
 * after the underlying request has been handled, until the application
 * programmer unrefs the object. */
class TtsRequest {
  async void pause ();
  async void resume ();
  async void cancel ();
```

```
802
803   /* The current state of the request. */
804   property TtsRequestState state;
805
806   /* The current progress of reading through the request, as an offset
807    * into the original text in Unicode characters. */
808   property unsigned int current_offset;
809
810   /* In a GLib API, these would be GObject::notify::state and
811    * GObject::notify::current_offset. */
812   signal notify_state (TtsRequestState state);
813   signal notify_current_offset (unsigned int current_offset);
814 }
815
816 enum TtsRequestState {
817   PREROLL,
818   PLAYING,
819   PAUSED,
820   FINISHED,
821   CANCELLED,
822 }
823
824 enum TtsPriority {
825   IMPORTANT,
826   MESSAGE,
827   TEXT,
828   NOTIFICATION,
829   PROGRESS,
830 }
831
832 enum TtsVoiceRate {
833   SLOW,
834   NORMAL,
835   FAST,
836 }
837
838 enum TtsVolume {
839   DEEMPHASIZED,
840   NORMAL,
841   EMPHASIZED,
842 }
843
844 enum TtsPitch
845 {
846   LOW,
847   NORMAL,
```

```
848     HIGH,
849   }
```