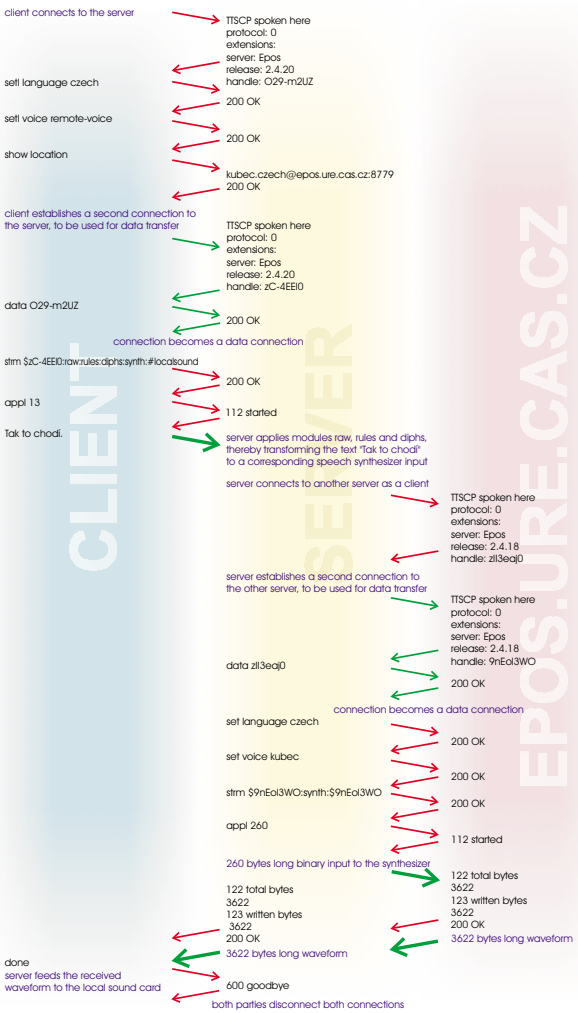
Sample TTSCP session using a remote speech synthesizer



TEXT-TO-SPEECH CONTROL PROTOCOL

Jiří Hanika
Faculty of Arts, Charles University, Prague
geo@cuni.cz

Petr Horák
Institute of Radio Engineering and Electronics,
Academy of Sciences, Prague
horak@ure.cas.cz

TTSCP is a client-server connection-oriented, both human- and machine-readable communication protocol, remotely similar to the File Transfer Protocol (RFC 959) in spirit.

TTSCP is offered as a standard interface for controlling generic speech processing applications, not only Text-To-Speech ones. It is primarily designed to run atop TCP or any other reliable connection-oriented underlying protocol.

TTSCP modules

Input and output modules

The following syntactic conventions hold for input and output modules (see the `strm` command in the diagram). If the module name begins with a `$`, the rest of the name is a data connection handle. If it begins with a slash, it is an absolute file name. Such absolute file names however form a name space distinct from that of the underlying operating system. In Epos, the name space is a single directory defined by the `pseudo_root_dir` option. It must be impossible to escape from such name space by inserting parent directory references in a file name or otherwise.

If the module name begins with a `#`, the rest of the name is a special input/output module identifier. The only identifier generally supported is `localsound`, which can only be used as an output module with the `waveform` type. Any waveform passed to this module should be played over using the local soundcard.

The output data type of an input module and the input data type of an output module are determined by the respective adjacent modules. If input and output modules are directly connected, it is assumed that the data is a plain text.

The TSR data type can not be sent or received, and may thus be totally implementation and architecture dependent.

Processing modules

At the moment there are only few modules implemented that do a real processing. All of them have fixed names and types.

Available processing modules:

NAME	INPUT FORMAT	OUTPUT FORMAT	NOTES
CHUNK	plain text	plain text	splits text
JOIN	plain text	plain text	joins texts
RAW	plain text	TSR	parses text
RULES	TSR	TSR	
PRINT	TSR	plain text	
DIPHS	TSR	speech segments	extract speech segments
SYNTH	speech segments	waveform	speech synthesis

`chunk`

The text is split into parts convenient for latter processing. These parts usually correspond at least to whole utterances; it is correct not to split the text at all, but care must be taken not to cause a split which significantly alters the final rendering of the text.

`join`

It is customary to use the `join` module just after a `chunk` module. If this module receives two consecutive texts such that the `chunk` module would not split their concatenation between them, the `join` module may merge them to a single text, that is, it may silently drop the first subtask and prepend the text to the text acquired later. This delay may cross the boundary of an `appl` command.

`raw`

The input text is converted in a language dependent way to the TSR, assuming it is a plain text without any specific TTS escape sequences or other special formatting conventions. Except for tokenization and whitespace reduction the conversion should not try to process the text, especially not in a language dependent way; this goal doesn't seem to be always feasible.

`rules`

The voice dependent TTS or other rules are applied to a TSR.

`print`

The TSR is converted to a plain text representation, suitable as a user-readable output. The conversion should be as straightforward as possible and should not emit any special formatting character sequences. Ideally, successive application of the `raw` and `print` modules should not significantly alter the text.

`diphs`

This module extracts the speech segment layer from the input TSR into the linear speech segment stream format; the rest of the TSR is discarded.

`synth`

The input speech segment stream is synthesized in a voice dependent way.

Explicit data type specifiers

Sometimes an ambiguity concerning the type of data passed at a certain point within the stream may occur. This is currently the case with streams consisting of input and output modules only (such as a stream to play out an audio icon from a waveform file to a sound card device); in the future, ambiguously typed versatile processing modules may be introduced, too. Sometimes the data type is semantically irrelevant (for example, a socket-to-socket forwarding stream), sometimes the default data type, that is, a plain text, is a reasonable choice. There are however instances where the type matters, like copying a waveform file to a sound card device: the waveform header must be stripped off and the appropriate `ioctl`s must be issued to replay the raw waveform data with the appropriate sampling frequency, sample size and so on.

The data types can be expressed explicitly by inserting a pseudo-module into the stream at the ambiguous position. Failing that, the output data type of the preceding module and/or the input data type decides the data type at this point. Failing even that, the server will assume plain text data.

TTSCP Commands

TTSCP commands are newline-terminated strings. Each of them begins with a command identifier, some of them may continue with optional or mandatory parameters, depending on the particular command. Each command generates one or more "replies", the last reply indicating completion and sometimes also some command-specific information.

`appl`

Apply the current data processing stream to some data. The parameter is a decimal number specifying the number of bytes to be processed.

`intr`

Interrupt an active stream. The parameter is a control connection handle and controls the connection to be interrupted.

The server should try to discard as much pending data as possible, including e.g. waveform data already written to a sound card.

The server will reply a 401 completion code to the interrupted connection, whereas a 200 completion code will acknowledge a successful `intr` command.

`data`

Turn this control connection into a data connection. The parameter is the handle of an existing control connection to attach this connection to. The sole consequence of this attachment relation is a disconnect of the data connection when the specified control connection is disconnected. (It is therefore common for a client to open two connections, to get their connection handles, to turn one into a data connection and to attach it to the other connection. That way the client obtains a control and a data connection which will gracefully shutdown even after the client abruptly disconnects.)

`delh`

Terminate a specified data connection. The parameter is the data connection handle to be terminated. If successful, the connection is disconnected by the server.

`done`

Issued as the last command in a session. The client may exit just after sending this command. The server should reply with error code 600.

`down`

Stop the server. Quit pending sessions. May disappear in the future.

`pass`

Attempts to validate an account, as given by a previous "user" command. If no valid "user" command was ever received, the internal server password may be used. This may enable some internal commands such as "down" or "setg". (Epos stores this internal password in /var/run/epos.pwd while it is listening on the standard TTSCP port.)

The password is a string of alphanumeric characters, dashes and underlines, no more than 250 bytes long.

`setl`

Set a server configuration parameter. The parameter is a whitespace-separated "option value" pair. The server may ignore this command altogether with an error code 442. In any case, this setting should never alter the execution environment of existing and/or future sessions. The server will reply with an error code 412 if the value assigned is illegal, or with 451 if the server is configured not to allow to change this parameter (may depend on the current authentication status).

The settings apply to the current session; use `setg` for more permanent settings. Note also that setting some options can have arbitrary side-effects.

If the option name is "language" or "voice", the command will attempt to switch the language or voice, respectively.

The standardization status of this command is still unclear. It is definitely reasonable to use compatible option names between server implementations where applicable, but the set of useful configuration parameters seems to be impossible to specify in advance. Any comment on this issue is welcome.

`show`

Show a configuration parameter value. The parameter is an option name. The server may ignore this command with error code 442.

`show languages` and `show voices` may be used for listing available languages, as well as available voices for the current language.

`strm`

Prepare a data flow stream. The parameter is a colon-separated sequence of data processing modules; commands such as `appl` cause specified data to be run through the modules from left to right. Any two adjacent modules must be compatible, that is the type of output produced by the one to the left must match the type of input processed by the one to the right. The leftmost module must designate a source (input) module for the whole stream, the rightmost one must designate a destination for the data produced by the stream.

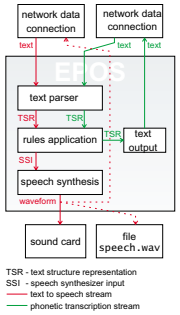
The stream is not automatically active. It processes data only when requested by the `appl` command.

The stream lasts until the next `strm` command or termination of the TTSCP connection, then it is deleted.

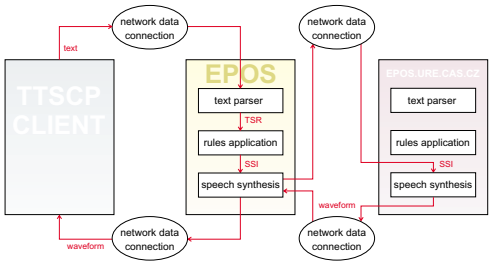
`user`

Should precede all TTSCP exchanges. Its parameter is "anonymous" or a local or configured user account name. Some other user names may acquire special meaning.

TTSCP STREAM EXAMPLES



SAMPLE TTSCP SESSION USING A REMOTE SPEECH SYNTHESIZER



This contribution is supported by Czech Ministry of Education project No 102/96/K087 and project COST 258. Further information can be found on *http://epos.ure.cas.cz/* .