TEXT TO SPEECH CONTROL PROTOCOL

Jiří Hanika, Petr Horák[*]

Faculty of Arts, Charles University, Prague, geo@ff.cuni.cz
[*]Institute of Radio Engineering and Electronics, Academy of Sciences, Prague, horak@ure.cas.cz
http://epos.ure.cas.cz/

ABSTRACT

The necessity of using the same TTS software within multiple operating environments, as well as separation of the TTS subsystem from the text source leads to a demand for a standard controlling interface. We describe our attempt at such standard controlling interface, the *Text-to-Speech Control Protocol* (TTSCP). The problems and the choices we have made for TTSCP as implemented within the framework of the recently introduced Epos Speech System are discussed. TTSCP is structured as a generic connection-oriented data stream processing protocol running atop the TCP protocol, and, while the control information exchange is human readable, it is tuned for efficient processing by server-side and both simple and complex client-side programs.

INTRODUCTION

In 1997, the TTS components used at the Institute of Radio Engineering, Academy of Sciences of the Czech Republic, Prague, and Institute of Phonetics, Charles University have been re-implemented and integrated into a language independent TTS system, which was later baptized Epos [1, 2, 3]. One consequence of the language independence is the necessity to read and parse a multitude of language dependent configuration files, including letter-to-sound and prosody modelling rules. We have decided to design the syntax of the configuration files as intuitive as possible, because their intended user would be a phonetician rather than a programmer, and not to precompile them into any machine-oriented form, for the same reason. That way, however, the start-up process of the system significantly exceeded the actual TTS processing in duration; for example, the configuration for the Czech and Slovak languages and several voices would have to be parsed, and then only a single sentence would be synthesized in a single voice. To avoid the delay, we decided to separate the TTS system (the *server*) from the utility program used to control it (the *client*). We have recognized that a standard TTS controlling interface was needed and in absence of such interface at that time we have designed the Text-to-Speech Control Protocol (TTSCP).

From the beginning, it was necessary for the protocol to be available under several operating systems, which ruled out most inter-process communication specifications as the underlying communication mechanism; the only generally available mechanisms have been the file system and networking, the latter of which is obviously superior in many crucial respects. We have chosen the TCP/IP networking suite over other networking standards, again because of its almost general availability. For reasons given later, TTSCP was destined to use a connection-oriented protocol, and to make it as simple as possible, it assumes a reliable underlying communication link; the simplest TCP/IP protocol with such properties is TCP, and this has been the final choice, because no higher level Internet protocols, such as the telnet protocol, seemed to add any useful functionality for the needs of the project.

The prevalent client type is another question, which had to be decided early. Many special-purpose communication protocols are machine-oriented, some are human oriented; most connection-oriented protocols fall somewhere in between. The primary use for TTSCP was clearly a routine communication between software components, and the human user was expected to access the facility indirectly. On the other hand, any machine-oriented approach must retain architecture independence anyway, and tends to raise speed versus versatility trade-offs, not to mention more difficult development or less flexible error reporting. For these reasons, TTSCP is optimized for efficient machine-to-machine communication, while the control information is a human-readable textual exchange, as it is customary with most connection-oriented Internet protocols, such as the File Transfer Protocol [6].

The protocol is based on a synchronous exchange of *commands*. Each command begins with a *command identifier* and a single space character, and ends with a line terminator sequence; any text in between serves as a *parameter* to the command. The client is allowed to send multiple commands at once, but the commands will be processed sequentially (though in parallel with requests issued simultaneously by other clients). At the point of completion of each command, a *completion reply* is sent to the client. Before a completion reply, the server may be sending an unlimited number of *intermediate replies* to signal progress and status information to the client. Each reply begins with a three digit *response code*, optionally a human readable rendering of the meaning of the response code, and a line terminator. Some intermediate replies are followed by additional information on a line by itself; such additional lines always begin with a single space character. To allow both simplistic and complex modes of client operation, the response codes are organized hierarchically: the first digit classifies the reply as intermediate (1), successful completion (2), error completion (4), connection termination (6), or permanent server shutdown (8). (See the enclosed exchange listings for examples; the commands are left-indented, whereas replies are right-indented.) The second digit is mainly useful with error completion codes, where it classifies the error condition and suggests a possible error recovery approach. The last digit serves for distinguishing between different error conditions from the same error class.

The overall communication model of TTSCP is not in fact TTS dependent. Rather, TTSCP is a general purpose data stream processing communication protocol, which could as well be easily extended to an Automatic Speech Recognition controlling interface, or to a dialogue system.

STREAMS

Speech processing software, even TTS technology alone, can be adapted for a variety of tasks; e.g. synthesizing individual machine generated utterances; interpretation of a contiguous stream of text; phonetic transcription; speech synthesis with a predefined prosody contour specification. Such tasks differ in the formats of input and output data, and in the set of operations to be applied to them. The available operations (transformations) are termed *modules*, and any particular sequence of them a *stream*. An experimental predecessor of TTSCP had predefined a fixed number of such tasks and corresponding commands have been used to invoke them; the resulting protocol however suffered from insufficient versatility, and the module binding has been changed from server driven to client driven. Hence in TTSCP, the client may interconnect the available processing modules in any order, provided that the data types used by adjacent modules are compatible, as explained later.

The protocol structure should encourage an effective implementation strategy. This includes technical issues such as an easy-to-parse syntax, but also general guidelines, such an avoiding unnecessary duplication of complex communication. One example of the latter principle can be seen with the typical TTSCP client session; every client uses the TTSCP server in a specialized way, and tends to repeat the same processing requests with different data during its lifetime. If every request would specify the sequence of modules to be interconnected and the detailed processing parameters for each of them, protocol parsing might introduce unnecessary delays before the actual processing of short bits of data. For this reason, the stream setup phase and the actual processing of data has to be separated in TTSCP, and thus the protocol has to be connection-oriented. A typical client then connects to a server, sets up a stream and a few operating parameters, and then it only issues simple data processing requests and supplies the data to be processed. Multiple simultaneous streams per a control connection are not supported to avoid a significant amount of cumbersome and rarely used protocol complexity; for example, an asynchronous status reporting scheme would have to be introduced with impact to almost every TTSCP command. Clients which require several fixed streams are expected to maintain a distinct control connection per stream, whereas clients of a very dynamical nature can create and use different streams within a single control connection sequentially.

Since the TTSCP control connection is text-oriented, some types of machine-readable data, like the speech waveform output, are not suitable for being transmitted e.g. as a parameter to a processing command. Also, inserting large amounts of input data in an arbitrary format into the TTSCP control connection would cause unwanted delays and errors in TTSCP parsing. To avoid this problem, all input and output data, which is not available to the server through the file system, is transmitted via a dedicated *data connection*. A data connection can be established by creating an additional control connection and issuing a specific command to change its status; after that, it may become the input or output component of a stream. The alternative, to establish data connections by connecting to a different TCP port, has been rejected, lest a server waste an additional reserved TCP port number.

DATA TRANSMISSION

The server is not expected to read a data connection actively except when instructed by a control connection (whose stream uses this data connection as its input). The control connection shall always specify the number of bytes to be processed by the server. That is because in speech processing, the client is often better informed about the optimal splitting of the text into individual *utterances*, and the server should respect the utterance chunking suggested by the client, without regard to the amount of data actually available for reading from a data connection. However, in other cases the client doesn't have any extra-textual information for a large text available, and the language dependent task of splitting the text somewhere has to be delegated to the server; utterance chunking (and utterance joining) TTSCP modules are therefore available at the server side and the client may choose to include them at the beginning of the processing stream and then feed the input text to the server in arbitrary-sized slices.

Likewise, the client typically doesn't actively try to read incoming data except when instructed to by the server. The server may generate multiple outputs (e.g. utterances) for a single input, and it signals the availability (and the size) of output with an intermediate reply. An ordering dilemma arises with this scheme which must be addressed. If the protocol requires the server to send the intermediate reply before the data itself is actually sent, a network or server failure may leave a client in an unusable state reading the announced data synchronously. Even worse, there is no way to interrupt the transmission at a user request. On the other hand, if the data may be announced only after is has been sent to the client, the protocol can deadlock: the server eventually becomes unable to send more data, because all available networking buffers are in use, and this prevents it from signalling the data to the client. These pessimistic scenarios arise only when the amount of output data is quite large, and the solution adopted by TTSCP is to use a two-level data transmission signalling. The server signals each successful *data transmission* (intermediate reply code 123), which is unrelated to the structure of the data sent and is used for flow control. In addition, the server also signals each successful *data processing task* completed (intermediate reply code 122) which is unrelated to the actual delivery of data, and is used by the client for gathering logical units of data. The data transmission signalling may never be invalidated ex post, contrary to the task completion signalling, which may be discarded by the server if an error condition occurs, or if the client uses a simultaneous TTSCP control connection to explicitly abort the operation in progress.

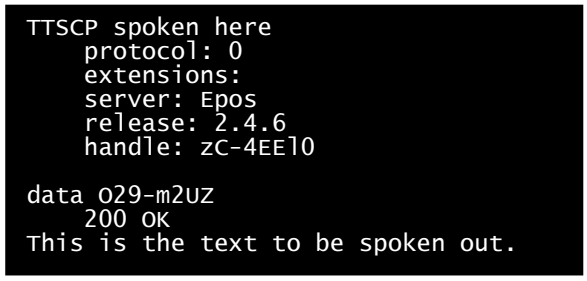

Fig. 1 Data connection establishment

The server closes a data connection if either the server detects a disconnection condition, or when its associated control connection is terminated, or when an explicit connection close request is received through any control connection.

OPERATING PARAMETERS

Speech processing is a complex task and a high level of configurability is essential for a versatile system. Often, simultaneous processing under different configuration environments is required, and thus TTSCP includes a command for setting such *operating parameters* (the `setl` command). The parameters to the command are *option identifier* and the *value* to be assigned to the option.

Any available options are classified as *general*, *language dependent*, and *voice dependent*. Once per a connection, a table of values of general options is maintained. For every language supported, a table of values of language dependent options is maintained, including a list of voices for a given language. For every voice supported, an analogous table of option values is maintained. When the client sets the *current language* (e.g. "`setl language slovak`", or using a markup included in the input data), the language dependent options start using the values assigned for Slovak, and the current voice switches to the current voice for Slovak. The voice can be switched in an analogous way.

TTSCP requires that the operating parameters for different TTSCP connections are independent, i.e. setting an option to a certain value using one connection doesn't affect the operation of any other simultaneous or future connection (unless the privileged `setg` command is used instead of `setl`). Conceptually, every new connection obtains a copy of all operating parameters (including all available languages and voices); but for efficiency reasons, the actual copying of any single table may be postponed until the first option of it has to be changed.

```
TTSCP spoken here
    protocol: 0
    extensions:
    server: Epos
    release: 2.4.6
    handle: O29-m2UZ

user user@host.domain.net
    452 user not found
set some_option on
    200 OK
strm $zC-
4EEl0:raw:rules:diphs:synth:/dev/dsp
    200 OK
appl 34
    112 started
    122 total bytes
     3622
    123 written bytes
     3622
    200 OK
done
    600 goodbye
```

Fig. 2 Control connection exchange

The interpretation of most options is server dependent; our implementation allows *boolean*, decimal *integer*, *string* and *enumerated* options. It is almost impossible to enumerate the set of all potential options for any speech processing system whatsoever, and thus operating parameters may become a weak place of TTSCP implementation interoperability, unless the efforts of its potential implementors are specially coordinated in this respect.

It is also understood that a portable client sets only the minimum number of necessary options and leaves the system default values where possible.

DATA TYPES

Every TTSCP module has associated a list of *data types*, which it is able to accept, and a list of possible output data types. The actual data types used are chosen from the intersection of the output and input types of the respective consecutive modules, or through an *explicit data type specification* within the stream setup command.

At present, the list of data formats understood by available TTSCP modules, in addition to the plain *text* and *waveform* (Microsoft `.wav`) format, is very limited. A simple *speech unit stream* which represents every segment of a spoken utterance with a speech unit number and the prosodic adjustments for pitch, duration and volume of the segment is used as the intermediate format between the text processing and speech signal processing modules. There is a major flaw this format, namely the impossibility of specifying multiple pitch points or volume points for a single segment without resorting to diphone splitting techniques. We are looking for a more flexible successor to this format; unfortunately, e.g. the MBROLA [5] `.pho` format supports multiple pitch points, but doesn't currently support volume adjustments at all. We are prepared to adopt it as soon as this issue is resolved. Another format, which we are going to support fully, is the SABLE markup language, the successor of STML [4], because the plain text format is not suitable for some machine generated texts, and includes no prosodic information.

Some modules operate over another, internal format: those, which are responsible for the actual linguistic aspects of the TTS processing, including prosody modelling. This internal format is not specified by TTSCP, and thus it cannot be directly transmitted through a TTSCP data connection, but parser and printer modules are available for converting the text data between the internal format and a standard plain text format.

JOB CONTROL

When the client is an interactive program, the sequential model of operation shows its limitations; it is often necessary to abort a command under processing, to suspend a task temporarily, or even to scan through a huge stream of text as it comes, speaking out only that much text as the time available allows, but still retaining a certain degree of intelligibility. Obviously, such level of sophistication requires the client to participate, but the protocol has to provide some basic asynchronous *job control* primitives to facilitate this.

The only job control command currently available in TTSCP is the *interrupt* (`intr`) command. The parameter to this command is a connection handle of an arbitrary control connection, which is currently executing a data processing command. The data processing command terminates unsuccessfully, and the error completion code indicates a user request as the reason; any unprocessed data is discarded.

A proposal exists for inclusion of *suspend* and *resume* commands into TTSCP, with the effect of temporarily suspending a data processing command, but a few semantic issues with the real-time TTS case have to be resolved first. Under certain hardware and operating system configurations, the data handed over to the system can no more be suspended, but either discarded or left for hardware processing; it is not clear which behavior should be preferred or even enforced by the TTSCP draft standard under such circumstances. Likewise, sound reasons are available for allowing the suspended control connection issue additional commands, but they are matched by reasons against such solution. Furthermore, the suspend command can consume memory resources for an unlimited amount of time, thus creating a potential for a simple permanent denial-of-service attack. For all of these reasons, we consider implementing the suspend and resume commands as a mere protocol extension, not a core feature to be included with every TTSCP compliant server implementation.

AUTHENTICATION AND SECURITY

To prevent data connection "hijacking", the protocol must ensure that a control connection, which attempts to plug a data connection to its own stream, is in fact the creator of the data connection. This is enforced by issuing a *connection handle* to every new connection; a connection handle is a randomly generated string of alphanumeric characters. The handle is thus only known to the creator of the connection; unrelated programs then can never plug the connection into their streams, as they cannot refer to it at all, but any trusted third parties may get the connection handle from its creator.

Hence there is usually no need to authenticate the client side, but in some circumstances the server administrator may limit the access to TTSCP services. Furthermore, some TTSCP commands (such as a server shutdown request) are not available to the ordinary user. For such purposes, the server associates a *user identification* with each control connection, and the *authentication status* (authenticated or unauthenticated). New connections are created with the "unauthenticated supervisor" identity. Commands are available for switching a user identity and authentication using a plain text password. (No challenge-response protocol is considered for TTSCP because the implicit connection handle authentication would have to be replaced by a more secure mechanism as well.)

The stream establishment command syntax allows the client to specify server-side files as the input or output to the stream, instead of data connections. This is the only interaction of TTSCP with the server-side file system, and it is therefore security sensitive. The server is however expected to create a separate, very restricted namespace for this purpose, which doesn't include all files actually accessible by the server program, but only audio icons, example inputs etc. The server is not required to allow the client to create new files this way, or even to implement this feature at all.

The exact impact of the authentication status on server behavior is not specified by TTSCP, but the basic communication model must always be observed correctly.

EXTENSIBILITY

TTSCP is still an evolving draft standard. The overall model of communication seems to be viable, but specific enhancements, such as additional modules, will no doubt be introduced, and some TTSCP servers may decide to implement only some of these, depending on their intended use. Though such considerations are still somewhat theoretical, protocol version compatibility measures have been introduced into the protocol early.

Immediately after accepting a TTSCP connection, the server sends a *session header* to the client. The session header includes, among other things, two protocol characteristics, the overall *protocol version number* and available *extensions*. If a client encounters a higher protocol version than its own, it should assume that the protocol is backwards compatible; if the server version number is higher by at most two, the extensions field is meaningful (otherwise a former extension may have been integrated into the higher protocol version and will not be announced explicitly). This allows reasonable backward compatibility while the protocol is being extended. Under normal conditions, the version number is raised only when the protocol is modified in a way which cannot be cleanly implemented as an optional extension.

In addition, the client is required to ignore unknown intermediate replies, and a name space is provided for custom (non-standard) TTSCP commands and modules.

CONCLUSIONS

This paper is based on the TTSCP draft standard, which is freely available as part of the on-line documentation for the Epos Speech System at http://epos.ure.cas.cz/. An implementation of TTSCP for UNIX type and MS Windows NT operating systems including a full source code is available at the same address. A few client applications of all levels of sophistication are also available.

We are very open to comments and discussion, especially from potential implementors of TTSCP servers or clients. The basic model of communication seems extensible also to non-TTS speech processing applications, including Automatic Speech Recognition, but any progress in this direction depends on feedback from experts in those fields.

REFERENCES

[1] Horák, P., Hanika, J.: User Configurable Text-to-Speech System. In: Proc.of the 8th Nat. Scientific Conf. with Int. Participation Radioelektronika'98, Brno, Czech Republic, April 28–29, 1998, pp. 212–215.

[2] Horák, P., Hanika, J.: Design of a Multilingual Speech Synthesis System. In: Sprach-kommunikation No. 152, 9. Konferenz "Elektronische Sprachsignalverarbeitung", Dresden, 31.8–2.9. 1998, pp. 127–128.

[3] Hanika, J., Horák, P: Epos – A New Approach to the Speech Synthesis. In: Proceedings of the First Workshop on Text, Speech and Dialogue – TSD'98, Brno, Czech Republic, September 23–26, 1998, pp. 51–54.

[4] Sproat, Taylor, Tanenblatt, Isard: A markup language for text-to-speech synthesis, Proc. of the 5th European Conference on Speech Communication and Technology, Rhodes 1997, ESCA.

[5] Dutoit, Pagel, Pierret, van der Vreken, Bataille: The MBROLA Project: Towards a Set of High-Quality Speech Synthesizers Free Of Use For Non-Commercial Purposes, Proc. ICSLP 96, Philadelphia.

[6] Postel, Reynolds: File Transfer Protocol, STD 9, RFC 959, USC/Information Sciences Institute, 1985.