MASTERTHESIS

The usage of model-based design for signal processing of an RTP stream

prepared at the
Salzburg University of Applied Sciences
Master Program
Information Technology & Systems Management

submitted by:
Tobias Kazmierczak

Head of Faculty:	FH-Prof. DI Dr. Gerhard Jöchtl
Supervisor:	DI Simon Kranzer

Salzburg, August 2012

Dedication

To my wonderful wife.

Affidavit

I hereby declare that I, Tobias Kazmierczak, born on June 27th, 1985 in Bensheim, wrote this thesis on my own and without the use of any other than the cited sources and tools and all explanations that I copied directly or in their sense are marked as such, as well as that the thesis has not yet been handed in neither in this nor in equal form at any other official commission.

Tenneck, August 30, 2012

Tobias Kazmierczak	1010581015
	Matrikelnummer

Acknowledgement

For the LORD gives wisdom:
out of his mouth comes knowledge.
Proverbs 2,6

I want to express my appreciation to Dr. Shuvra Bhattacharyya for introducing me to an exciting area of digital signal processing and offering me an excellent research topic for my master thesis. I want to thank my supervisors Dr. Chung-Ching Shen who continually accompanied me during this scientific research process. Without his persistent guidance and support this master thesis would not have been possible. Additional thanks to my colleagues Lai-Huei Wang, George Zaki, William Plishker, Zheng Zhou and Hsiang-Huang Wu for advice, discussions and time.

Special thanks to Simon Kranzer, my supervisor at my home University, for encouraging me through the whole process of preparing and writing my Master thesis. Your continued feedback and your time supported me immensely.

I would like to thank my parents, my cousin Sarah, my sister in law Anna and especially my wife Julia for their support to help me reach this point.

Thanks to everyone who made it possible for me to achieve personal and academical goals: my parents for their advice when needed and especially Judy for being the greatest sister. My employer Eurofunk Kappacher who allowed me to take a long time off to spend the necessary abroad. Also, the hospital Schwarzach to allow my wife to pause her job to accompany me.

In addition, I want to thank the International Office of the University of Applied Sciences in Salzburg, Austria and the Austrian Marshall Plan Foundation for making it possible to get involved with the DSPCAD Research Group hosted by the University of Maryland, USA. Special thanks also to Inkeun and Youkyung for swapping houses with us.

Common information

First and last name:	Tobias Kazmierczak
Institution:	University of Applied Sciences Salzburg
Curriculum:	Information Technology and Systems Management
Title of thesis:	The usage of model-based design for signal processing of an RTP stream
Keywords:	Dataflow, Signal Processing
Supervisor at University:	DI Simon Kranzer
Head of Research Inst.:	S. S. Bhattacharyya

Abstract

The dataflow scheme is a widely used programming method for digital signal processors and has many advantages for signal processing applications. This master thesis describes the use of dataflows for implementing voice data transmission with different Real-time Transport Protocol (RTP) algorithms, mainly packet loss concealment, voice activity detection and comfort noise generation. These algorithms require real-time processing capabilities and have an unpredictable behaviour. To model these applications a non-deterministic dataflow has to be used. On the basis of enable invoke dataflow an API has been derived from the lightweight dataflow to implement the behaviour of RTP algorithms. The encapsulation of signal processing in the actors allows the designer to specify the token flow and enables the low-level programmer to concentrate on the development. The modelling is done in a generic way that different implementations of RTP algorithms can be adapted to the model.

Contents

List of Figures

List of Tables

Listings

1

Introduction

The world around us is becoming more and more connected. Humans in the developed countries are surrounded by electronic devices which are produced by different companies around the globe. Experts say, that as "electronic systems become smaller and more complex, it is becoming increasingly difficult for manufacturers to efficiently develop high-performance embedded systems." [26]. Those embedded systems have to perform critical tasks like picture a tachometer on a digital display in a BMW 7 series. Therefore, the factor of stability plays an important role in the development and testing of embedded devices. And additional requirements need to be fulfilled: software has to be optimized both on a low and high level. At the same time the simplification and encapsulation of complex systems is a large challenge. As time continues to be a competitive advantage, rapid prototyping and development are key factors to survival and success.

1.1 Model-based Design

Before a programmer starts to develop source code for a program, he begins to plan the structure of the application. He normally uses a domain model to create a concept for a specific problem. It describes entities, their attributes and the relationships between the entities.

However, a computer execute machine code. Depending on the type of the processor, the machine code varies. Processors are often made for special purposes, such as

signal processing. Especially in the beginning of the computer age computers were programmed with assembly language. The human readable code contains instructions like copy, add, multiply or jump to another instruction. With a combination of these instructions, useful programs can be put together. A programmer has not only have to think about what he wants to accomplish but mostly about memory, overflow and the correct combination of the instructions.

Even today, low-cost digital signal processing applications are programmed using the assembly language. Applications on digital signal processors (DSP) have to be very efficient because of limited power and resources. Using languages which support object-oriented programming would create too much overhead due to using extra layers and special features. To benefit from advantages of model-based design, another approach for DSP applications has to be considered: dataflow modelling.

Dataflow modelling is widely used in the area of DSP application development. The concept of dataflows with its actors and edges is very closely connected to the representation in block diagrams. Those diagrams usually describe algorithms in signal processing. Another advantage of dataflows is a shared development effort between the actor programmer, who implements the actual processing (e.g. inverting, filtering, ...) and the application designer, who uses actors and connects it with edges to create an actual application.

The dataflow approach is used in this thesis to benefit from the advantages of model-based design and low level implementation for signal processing applications.

1.2 Communication over IP networks

The history of audio transmission shows that the usage of Voice over IP (VoIP) is a relatively recent development. Even if the resulting audio stream seems similar, the concept has radically changed. The connection in a public switched telephone network (PSTN) is used for voice only but the virtual VoIP connection is transferred over the World Wide Web (WWW).

This introduced a whole new dimension of applications to manage the complexity of

packet switching to match the features of the traditional method of line switching. A part of these applications will be implemented in the concept of dataflows. Since audio transmissions over the Real-time Transport Protocol (RTP) are often pictured in block diagrams it fits well into the context of model-based design.

1.3 Purpose of the Research

The first challenge is to determine if it is possible to model highly dynamic RTP applications with dataflows and what kind of dataflow model will cover the requirements. A main purpose of the research is to discover possible improvements due to the usage of dataflows in the processing of RTP streams. Is it possible that the modelling of applications with dataflows will show effects on quality of implementation and enhance testing strategies? These are only some questions which will be covered in this thesis.

1.4 Limitations of the Research

Using RTP to send and receive audio content does only include the transmission of samples and timing information. Additional information about jitter and payloads are transferred through the RTP Control Protocol (RTCP) which is further described in the RFC 3550 [47]. For the initialisation of RTP sessions other protocols have to be used. Examples would be H.232 or the Session Initiation Protocol (SIP). The dataflow implementation will include none of the protocols named above.

2

Audio transmission optimization for RTP Streams

As mentioned in the introduction, transmitting a continuous data stream across a packet orientated network can cause various problems. Every data packet might be routed differently, therefore sequential arrangement might change (see figure on page 185 of [38]). RTP enables the transport of isochronous data; this means the timing of a data packet is critical to the usage. An audio stream is a good example for this kind of data. Any delay of a packet will be audible. To compensate delays, a jitter buffer is used, which allows to compensate the jitter of the network. Over a certain time, usually 20 to 100 ms, the packets will be brought into the right order. This allows the isochronous data to have the exact same timing as they had when they were captured. In a world-wide network packets are not only reordered, sometimes they never get to their destination or the jitter buffer deadline is reached, before the packet arrives. If the packet contains 20 ms of audio data, it cannot be issued. In case of a bad IP network connection this error will have audible effects on the communication. Therefore, audio transmission optimization has to especially deal with the case of lost packages [32, 17].

A telephone call offers a bidirectional communication. However, in a normal conversation only one person speaks at a time. Theoretically this would mean that 50% of the bandwidth could be saved. Considering periods of silence, only about 40% of the data packets are containing meaningful information. This allows a massive saving in bandwidth during a telephone call which is transacted over an RTP Stream [29].

This chapter gives a short insight into audio transmission over an RTP stream and will show how to deal with lost packages and insignificant data.

2.1 The Real-time Transport Protocol

In January 1996, RTP was first published by the Internet Engineering Task Force. Its goal is to deliver video and audio streams in real-time over the Internet Protocol. It is used in applications such as VoIP calls, video conferencing and video-on-demand. Supporting the multicast delivery method it also enables Internet television. RTP is an application layer protocol and usually based on the stateless User Datagram Protocol (UDP). Figure 2.1 shows the structure of an RTP packet [47, 46].

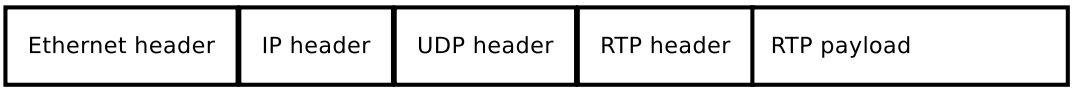


Figure 2.1: Format of an RTP Packet

Most packages which are traveling across the Internet use the Transmission Control Protocol (TCP). This protocol ensures the correct delivery of a data packet. If a package gets lost or is corrupted, it will be repeated after a certain time. Although the framing method for RTP is defined for a TCP connection in [33], it is not in widespread use. The reason for that is the fact that after 80-100 ms (depends on the size of the jitter buffer) a packet is useless because the stream is already played. The TCP timeout for the retransmission of a packet is usually 5 seconds which is far more than the jitter buffer [33].

Connection oriented transmission with TCP contrasts the connectionless communication via IP/UDP. No logical or physical connection is established which ensures the transmission of all data over the same path. Therefore, it is described as stateless. No message flow is defined by the protocol, this means every packet is individually transferred over the network. This can lead to different delays and is discussed in section 2.2.1. The header of an RTP packet is an important part and the first step to enable real-time transmission. It will be described in the following section [31].

2.1.1 Header

Parts of the RTP header which are important to understand audio streaming applications will be described in this paragraph. For further details read [47]. For implementing the examples in chapter 4, only parts of the header are critical for the functionality of the algorithms. Other parts will stay unassigned and could be covered through additional components or the usage of a library, e.g. oRTP[1]. Therefore, only parts of the header will be shortly described. The minimum size of 12 bytes of an RTP header are visualized in table 2.1.

bit offset	0-1	2	3	4-7	8	9-15	16-31
0	Version	Padding	Extension	CSRC count	M	PT	SN
32	Timestamp						
64	SSRC Identifier						
96	n CSRC [n = 0...15]						

Table 2.1: RTP Packet Header (adapted from [18])

The marker bit (M) labels special events. Such a special event can be the first package of active speech when using Voice Activity Detection (VAD).

RTP offers different kind of payload types (PT), which can be categorised in three classes: audio (A), video (V) and both audio and video (AV). Payload types 1-23 are audio encodings, which contain the most common payload types (PCMU, PCMA, GSM, etc.). To extend the number of possible payload types, more codecs can be assigned dynamically as payload. Examples are G.726, G.729 in their different variations. The dynamic payload type is defined through the conference control protocol, more information is available in [28]. Unknown payload types (see section 2.1.2) are dropped by the receiver. The payload is seven bits long and is located in the first 32 bits of the header [47, 46].

Also the sequence number (SN) which is two bytes long belongs to the first 32 bits. This number increments with each packet. When the maximum is reached, the number will be wrapped around. This happens about every 20 minutes in a VoIP call with a sample

[1] An open source project, written in C originally for the free SIP VOIP client linphone [37]

time of 20 ms. The initial value should be random to make attacks more difficult. The same should be true for the timestamp, that fills the next 32 bits of the header. At a fixed rate of the audio stream, timestamp increments by the length of a frame. The timestamp will be used for package synchronization in the jitter buffer of the receiver. Therefore, it has to be increased for every packet, even if it is dropped because of silence when using a VAD algorithm [47, 38].

2.1.2 Payload

The payload contains the actual data and forms, together with the header (see section 2.1.1), the RTP Packet. This can be compressed video data or audio samples. The type has to be declared in the payload type of the header. The most common payload type for VoIP is PCMU/PCMA (G.711) data. It is a 8 bit sample which have been compressed by a non linear characteristic curve. This curve is slightly different for applications in USA/Japan (μ-law) and Europe (a-law). Besides linear encoding with 16 bits, G.722 codec is getting more important for so called HD-VOIP. Instead of the common telephone frequencies of 300 Hz - 3400 Hz, the G.722 codec transmits frequencies of 50 up to 7000 Hz. This results in a higher quality while using the same bandwidth. The disadvantage is that it needs more processing power both on sender's and receiver's side.

2.2 Packet-based transmission problems

As described at the beginning of this chapter, using a VoIP infrastructure instead of a PSTN will cause new kinds of problems. A solution exists for most of the matters:

- Latency -> enhanced by QoS Networks
- Jitter -> Jitterbuffer
- Packet loss -> Packet Loss Concealment (PLC) Algorithms

Not to deal with these issues leads to a poor experience with VoIP technology. Figure 2.2) shows causes for voice quality issues [2]. The following sections give more details on the problems which appear on packet-oriented networks and explain possible solutions.

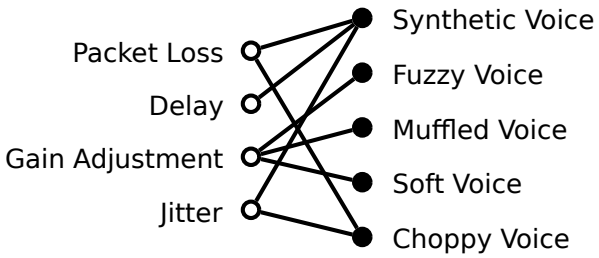


Figure 2.2: Causes of voice quality issues (adapted from [2])

2.2.1 Latency

Latency is the time that passes by between the voice is digitalized at the senders side and comes out of the speaker at the receivers side. This delay comes mainly from three different sources, which have more or less influence according to the basic parameters and conditions.

The first delay is caused by serialisation. As pointed out in the beginning of chapter 2.1, voice is segmented into packages which are called frames. If the length of a frame is 20 ms, the receiver cannot reproduce the first milli-seconds of the packet until the whole packet arrives. This delay can be shortened, when using smaller package sizes.

The second part of the latency on VoIP connections is called propagation delay. It is caused through physical limitations of the transport of information between point A and B. Using a fiber optic cable, the electrons travel with about 125k miles per second. For the distance from Washington D.C. to Los Angeles (2500 miles) an Internet packet needs about 20 ms. It can be even longer because the shortest way is not always the cheapest. Packet switching and routing times are not included in the propagation delay[2].

A third delay that causes latency is the processing delay. It is additional to the time for compression, packetization, decoding and the different algorithms, that are applied to the audio stream during transmission. Using the G.722 Coding (see 2.1.2) adds up to 3 ms of delay, which includes look-ahead and processing time. In comparison to that, G.711 adds a few frames (300-500 μs) to the processing delay. Every network device that forwards the frame adds a delay [17].

2.2.2 Jitter

Theoretically every RTP package that travels from sender to receiver can use different path in the WWW. Therefore, the major part of the latency is not static because the propagation delay and the processing delay can vary. To compensate this, a VoIP endpoint is equipped with a jitter buffer. A static size may add unnecessary delay to an RTP stream or miss a high rate of late packages. Therefore, a dynamic jitter buffer is widely used. This dynamic buffer counts the number of late packages and creates a ratio to the packages that arrive in a certain time period. When the ratio increases, this time period will be increased. The usage of this algorithm will cause a minimal delay at all times. Experience shows that a jitter buffer of 60 ms is an appropriate value [2, 14, 42].

Jitter has a great influence on the voice quality. Any late packet will end up in silence, if it is not concealed. Already simple algorithms can improve the audio quality if packets are not delivered in time. Two different approaches will be considered later in this chapter, see 2.3. Also quality of service (QoS) mechanisms can have a positive influence on the jitter [14].

2.2.3 Packet loss

Packet loss is a characteristic of a real world IP network. Different reasons exist for the loss of packages, some of them are depending on the network infrastructure. A frequent reason is the bottleneck between two networks, for example between a Local Area Network (LAN) and the Wide Area Network (WAN). Heavy traffic causes the drop of IP packages. Thereby the net transmission rate of a TCP connection decreases through the repetition of lost packages. The result will be the same in return for the error correction. As described in the beginning of this chapter, the same behavior on a UDP connection will have a major impact on the quality of an UDP connection [3].

In a real world network the loss of packages can have different kinds of distributions. A lot of speech quality tests use the stochastic model of Bernoulli to describe the randomness of packet loss. Usually this is not the case when the cause of the loss is

considered. Often it happens in times of a capacity overload of a network. Therefore, packet loss is described as “bursty loss”, rather than as “random loss”. This can be described in different models, e.g. the 2-state Markov loss model. The two transmission probabilities p and q define the probability for a change of the state. To reach a higher accuracy more states are required. Considering these models together with the reasons for packet loss will help choose techniques and parameters when dealing with different algorithms [2].

2.3 Dealing with Packet Loss

Not dealing with packet loss will lead to a choppy sound of the transmitted audio. It also leads to clipping, which will be noticed as a skip in the conversation. Two basic approaches exist to fill the packet gaps with substitute packets. One technique, called PLC, is applied only at the receiver side. It compensates lost packets based on information from previous packets. More enhanced methods can give the impression of a continuous signal, although the speech quality is lower compared to an error free signal. This technique only adds a short processing delay. Another approach are Forward Error Correction (FEC) and Low-Bitrate Redundancy (LBR). Both are using redundant data streams to compensate lost packages. They have to be applied on the senders and receivers side and the same technique has to be used on both sides. Using a redundant stream will lead to better results than PLC, but in most cases the costs are higher than the benefit, depending on the network infrastructure. It has to be considered, that both FEC and LBR increase the network traffic, therefore also the network delay. This master thesis will only deal with PLC [39, 44, 34, 2].

2.3.1 Simple Packet Loss Concealment

The goal of an PLC algorithm is to hide transmission loss on the receivers side. The most simple algorithm to implement a PLC is to store the last package and repeat it, if a packet is missing. Other than total silence, the receiver will hear the same frame again. On short frames (e.g. 5 ms) this could be not noticed at all as long as only

one frame is missing. On streams with 20 ms length, the voice may sound synthetic and choppy. Multiple repetitions of a frame will result in a corrupted audio signal. Therefore, the packet is only repeated once or twice and cannot cover a longer packet loss. This method is explained, even though it is not used in VoIP communication to show the difference to another PLC algorithm. In this example every 5th frame is dropped and the last frame is filled in again. Looking at the audio graph (figure 2.3) it can be seen, that the 10 ms sample from 0.100s to 0.110s is repeated at 0.110s unto 0,120s.

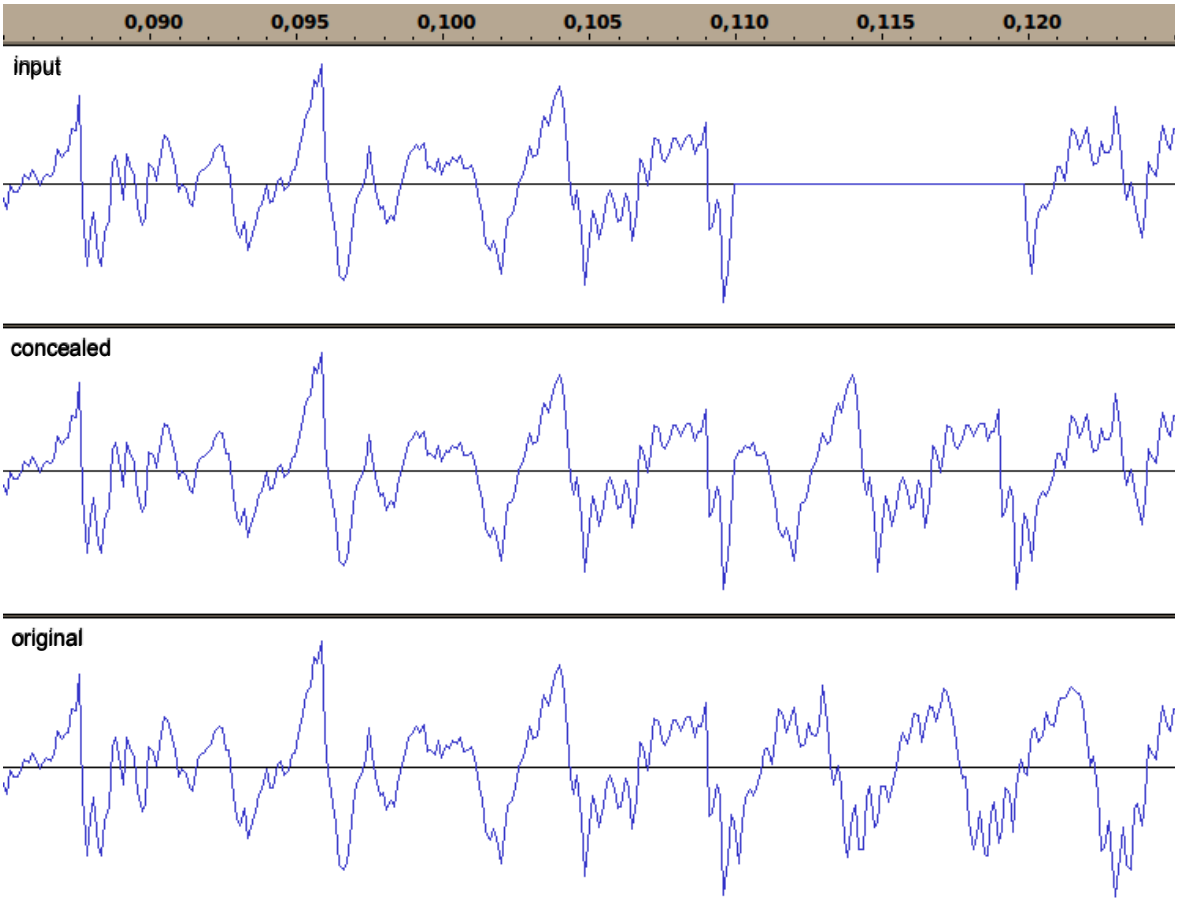


Figure 2.3: Concealed packet loss from 0,100s to 1,120s (with simple repeat)

A major drawback using this algorithm is the jump at the beginning and the end of the frame. This is not always visible but audible most of the time.

2.3.2 PLC as recommended in G.711

The International Telecommunication Union (ITU) recommends different PLC algorithms as part of their speech coders standards G.711, G.723.1, G.728 and G.729. In this thesis the recommendation in G.711 will be used because the implementation in C is already given and the algorithm is easily understandable. The following paragraphs will briefly describe the algorithm of the recommendation G.711. The algorithm is based on audio data sampled with 8 kHz. The audio packages have a size of 80 samples, which correspondents 10 ms. The ideal result would be a synthesized packet that will have the same characteristics as the missing audio signal in the package. Achieving this is not an impossible task, because speech signals are often locally stationary. The only condition for an almost inaudible concealment is a smooth signal. Rapid change in the signal causes a problem in every PLC algorithm [23].

To weaken the effect of a rapid signal change, the output of the algorithm is delayed by 3.75 ms (30 samples). This makes it possible that the synthesized signal is faded into the good signal after a packet loss. This delay is also used to create an overlap at the beginning of the loss. To create the synthesized signal, the history of the audio samples is saved in a history buffer. The length of this buffer is almost 5 packages long: 48.75 ms (390 samples). It is organized as a ring buffer and filled with the payload of every incoming packet [23].

On the detection of the missing frame the whole history buffer is copied to the pitch buffer which will be the relevant buffer for the duration of the erasure. The most recent 20 ms and tabs from 5 ms to 15 ms are used to estimate the pitch period by finding the peak of the cross-correlation. The synthesized signal is created of one or more pitch periods and the overlap is done with 1/4 wavelength of the pitch period and the original signal [23, 27].

To avoid unnatural harmonic artifacts (beeps) on the concealment of further frames, another slightly different action is used: multiple pitch periods are used to reduce these artifacts and increase the variation in the signal: two pitch periods are used on the second loss and three for even more losses. The length of the overlap stays at the original 1/4 wavelength. This signal is stored in an independent “lastq” buffer [23].

With each concealed packet the payload will get attenuated. After five packages the signal will be zero. The overlap function insures that the transition is smooth when a good packet arrives. This overlap period can last from a quarter pitch wavelength up to the full packet size of 10 ms. Figure 2.4 shows an example with the input signal at the top, the concealed in the middle and the original signal at the bottom. In order for the signals to be in sync the original signal has be also delayed by 3.75 ms. [23].

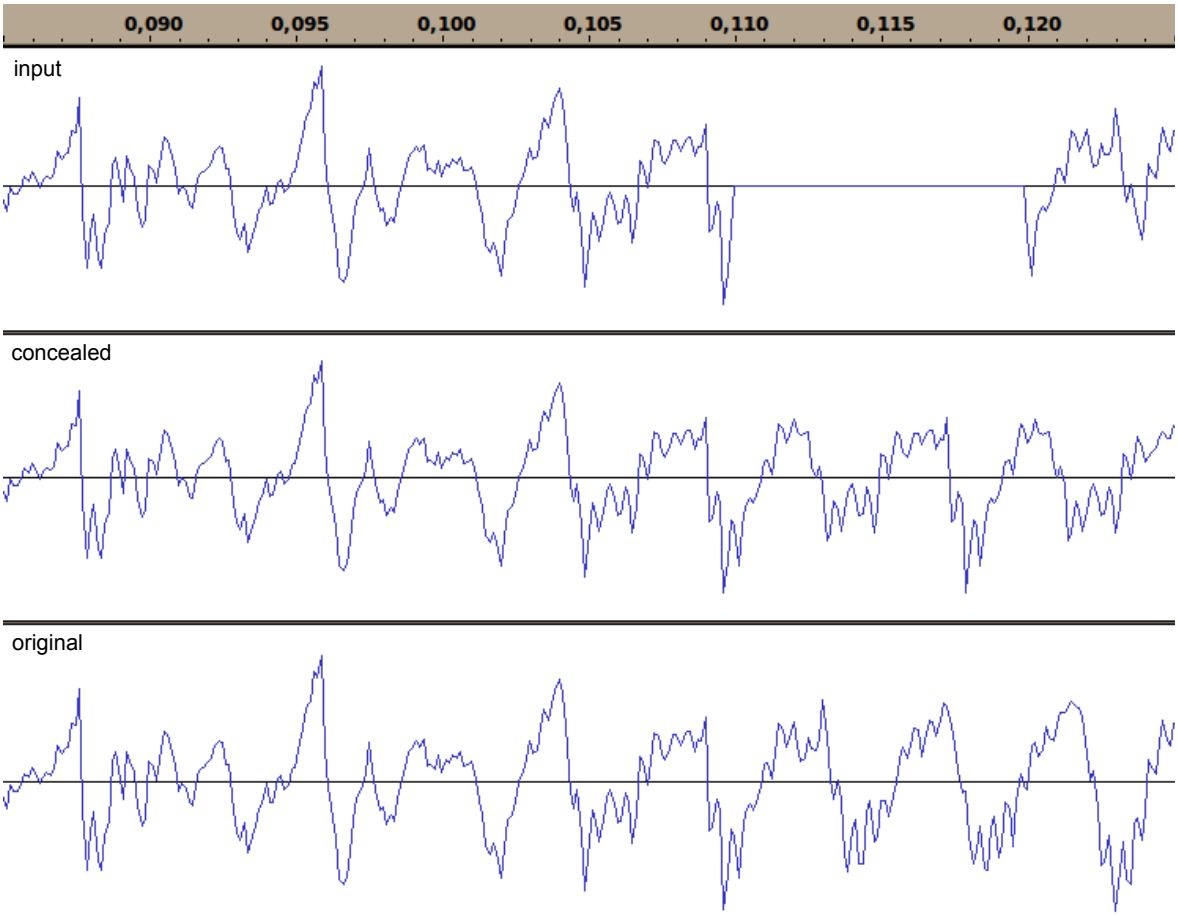


Figure 2.4: Concealed packet loss from 0,110s to 0,120s (with PLC G.711 rec.)

If compared to the simple PLC (see figure 2.3) the waveform in figure 2.4 shows a smoother and more accurate synthesized signal.

2.4 Saving transmission Bandwidth

Over the last years prices for IP traffic dropped and the capacities of the WWW are constantly enhanced [24]. Better codecs have been developed to minimize the bit rate of an audio connection. But there is still more potential: a bidirectional telephone connection carries relevant information in only about 40% of the time as described in the introduction of the chapter. Therefore, an algorithm has to recognize the periods of silence or noice without relevant information and needs to change the output of the sender. The ITU-T offers several recommendations for codecs like G.711, G.722, G.726, G.727, G.728 and G.729. The algorithms will be explained as described in G.729 Appendix B because it is presenting the most detailed description and includes the implementation. A block diagram is presented in figure 2.5 for a better understanding of the interaction of the algorithms which will be described in the following sections [22].

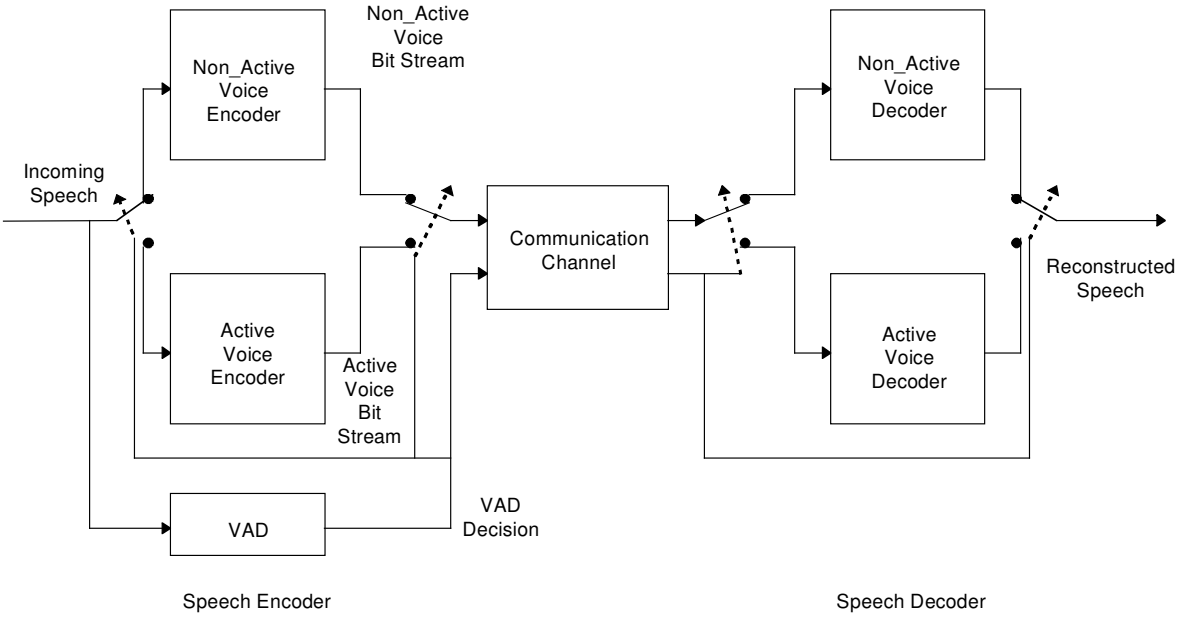


Figure 2.5: Block diagramm with VAD and CNG (from [22])

Incoming speech packets are analyzed by a VAD algorithm and a decision is made: active or non-active voice. Different blocks are used to encode the RTP packages containing the audio data. Together with the result of the VAD decision they are transferred through a communication channel. One the receivers side, the speech is

reconstructed: active voice, which has been encoded with G.729 is decoded and noise parameters are converted into synthesized noise. Outgoing speech is not completely similar to the incoming speech but is transferred with less bandwidth [22].

2.4.1 Voice Activity Detection

A voice activity decision is made for every RTP packet and the result can eighter be “0” or “1” representing relevant information or silence (active or non-active speech). Incoming audio frames are processed in order. Four characteristic features are extracted from the input signal:

1. Full frame energy
2. Low-band frame energy
3. A set of Line Spectral Frequencies (LFS)
4. Frame zero crossing rate

Formulas to calculate these values can be found in the recommendation [22]. The initial decision is based on the frame energy, a value above 15dB results in a decision for active voice. The next step is a calculation of differences between the current parameters and the averages of the background noise characteristics. A multi-boundary VAD decision is used to consider the current frame voice or non-voice. Altogether, 14 boundaries defined are in the four-dimensional space. The energy level is factored in to allow a smooth decision between frames. To provide a correct decision in case of noise, the algorithm updates the adaptive background noise energy threshold. This is also known as signal-to-noise threshold. All details can be found in [22, 2].

2.4.2 Payload for Comfort Noise

Total silence in a period of a negative VAD decision leads to a bad user experience. It is natural to hear a little noise, therefore the receiver will have the impression of an unstable connection or disconnection. A solution can be the transfer of certain

characteristics of the suppressed noise. This is done by a defined payload, known as the Silence Insertion Descriptor (SID) frame. It consists of a noise level and optional spectral information in the form of reflection coefficients. The number of coefficients which are stored in octets is left unspecified and depends on the type of application and the complexity level of the Comfort Noise Generation (CNG) algorithm [55].

The noise level is specified in -dBov encapsulated in the first byte of the SID. The most significant bit (MSB) is not used and is always set to "0", therefore values are in the range of 0 to -127 using dBov. The unit is defined as "the level relative to the overload of the system [55]. Further octets are filled with relative spectral information which can have values between -1 and 1 and are quantized with 8 bit in the range of 0 to 254. The quantized values of the reflection coefficients (k_i) are calculated using equation 2.1 ($N_i = 0 \ldots 255; -1 < k_i < 1$) [55].

$$k_i(N_i) = \frac{258 * (N_i - 127)}{32768} \tag{2.1}$$

The content of the CNG payload starts with the level (one byte) as described above and is followed by the m coefficients (m is the order) which results in the total length of $n = m + 1$ [55].

Using a SID frame the payload type in the header has to be set correctly. A static payload type exists for a clock rate of 8 kHz which has the number 13. Other rates have to use a dynamic payload. The marker bit is not set in RTP packets with a CNG payload. It needs to be set with the first transmission of a voice packet. If multiple channels are transferred over an RTP packet, the CNG payload has to have the same settings for every channel [55].

The recommendation for the G.729 implementation uses a discontinuous transmission (DTX) algorithm to transfer only CNG payloads which contain new information. This is achieved by using absolute and adaptive thresholds and results in a minimal transmission rate for SID frames [22, 55].

2.4.3 Comfort Noise Generation

The frame signal level and the reflection coefficients from the SID frame are used to generate the comfort noise. This is done by introducing an excitation signal into Linear Predictive Coding (LPC) filters. This excitation is a mixture between the characteristics of the actual signal and a white Gaussian noise. The calculations are done in the encoder and the decoder, and the difference to the previous signal is transferred in the SID frame. More details can be found in the G.729 Annex II [22].

3

Dataflow Computational Model

A model of computation is defined by the semantics of a language. It describes the interaction between modules abstracted from the material details of a device. The well known "von Neumann model" can be characterized by the strict sequential procedure of instructions. Languages like C define step-by-step computations which form an application. In 1972, J. B. Dennis engineered the concept of dataflows which fundamentally differs from the "von Neumann model". It is the "representation of the logical scheme [...] in which the sequencing of function [...] and the flow of values [...] are specified together" [13]. A function is free to proceed as soon as the value is ready which is required by the next function to proceed [15, 43].

In the paradigm of dataflow modelling, a program is defined as a directed graph G. It contains of a combination of vertices and edges: $G = (V, E)$. Computations in this graph are done by actors (vertices) and data is transferred in FIFO's which are representing edges. Data is designated as tokens and transferred from an output of one actor to the input of another. Source actors (actors with no input edges) model input data for a dataflow graph, and sink actors (actors with no output edges) model output interfaces (e.g. writing to a file or sending data to a digital-to-analog converter). A dataflow graph is data-driven which means an actor can only be executed (fire) when enough data (token) is available. An actor can have multiple inputs and outputs, therefore all input edges have to have a sufficient number of tokens. When firing, a defined number of data is consumed (consumption rate), the computation is executed and tokens are produced to fill the output edges (production rate). In this model there

is no specific order determined in which actors fire. It depends on the hardware or the compiler [5, 19].

3.1 Types of Dataflow Models

Dataflow models can be divided into two major groups: static and dynamic. In a static dataflow the number of transferred tokens is constant and known at compile time. This results in a completely deterministic behaviour that cannot vary on the bases of input data. It can be expressed with

$$t(p, i) \tag{3.1}$$

where t is the constant transfer rate of tokens for all actor ports p and all invocation indices (positive integers) i. This contrasts the dynamic dataflow: dynamic data rates for consumption and production can be handled, therefore, scheduling of dynamic graphs requires special interest. This includes retaining efficiency and keeping predictability. The following paragraphs is going to give a short overview over certain types of dataflows and their fields of application [5, 40, 50].

3.1.1 Synchronous Dataflow

Synchronous dataflows (SDF) belong to the group of static dataflows and are the most popular used in DSP system design. The amount of consumed and produced tokens is constant which benefits in compile-time predictability. This allows static scheduling, memory determination and deadlock detection. It opens possibilities for a variety of optimizations and resources can be used very efficiently. The handling is very straightforward for developers whereas flexibility is limited. Figure 3.1 shows a simple SDF graph with the consumption resp. production rate right next to the actors [40, 50].

Cyclo-static dataflows (CSDF) [40, 50] are a common extension of SDF. The consumption resp. production rate can vary between two firings as long as the variations follow a periodic pattern. This change of rate is implemented by phases which are executed

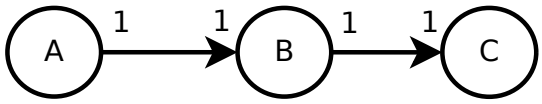


Figure 3.1: Simple SDF graph

on every firing. This can realize a change of consumption rate as 1, 3, 0, 1, 3, 0,... on an input edge. It allows static periodic schedules and compilation as a cyclic pattern of a SDF graph [4].

3.1.2 Parameterized Synchronous Dataflow

Parameterized synchronous dataflow (PSDF) is an integration of SDF in the meta-modelling framework of parameterized dataflows (PDF) [4]. Thereby it is transformed into a dynamic dataflow while preserving useful advantages of SDF. PSDF actors are defined through a set of parameters which control the behavior. This allows a change of consumption and production rate after each firing without restrictions. Edges can be configured in a similar way. Each actor has a subsystem which consists of an init, a subinit and a body graph as shown in figure 3.2 [45].

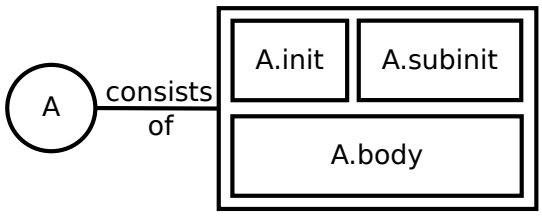


Figure 3.2: Simple PSDF actor (adapted from [45])

Init and subinit graph control the body graph behavior. This allows adjustments at run time on the basis of input data. The init function can be initialized with a start value and one edge could be connected to the subinit graph, setting relevant data on each firing. The body can receive a token to process calculations on it [4].

3.1.3 Enable Invoke Dataflow

A recently developed dataflow model is called enable invoke dataflow (EIDF) [40]. Its name results of the division of actors into enable method and invoke method. The model allows application design with a structured dynamic behaviour which meets conditions of dataflows which are described in the beginning of chapter 3. Every actor can have a set of modes and can be executed in one of them when invoked. Each mode has a constant consumption and production rate of tokens. This provides a structure while mode switches allow dynamic behavior at run time. Before an actor is invoked it gets enabled and checks the availability of input and output FIFOs. If the function returns true the actor will get invoked, processes data and defines a "next mode" for deterministic applications or a set of valid "next modes" for non-deterministic applications. "Next mode" means an actor is processed in this mode on the next execution.

Based on the definition of a dataflow the enable function (ϵ) of an actor $a \in V$ can be defined as (3.2) [40]:

$$\epsilon_a : (T_a \times M_a) \rightarrow B \tag{3.2}$$

where T_a is a tuple of tokens at each input and M_a is a set of valid modes. The return value is B which can be true or false. Calculations in the enable function should only include the count of tokens in the buffer to avoid redundancy with the calculations in the invoke function. This function ι for actor a can be defined as (3.3) [40]:

$$\iota_a : (I_a \times M_a) \rightarrow (O_a \times Pow(M_a)) \tag{3.3}$$

where $I_a = X_1 \times X_2 \times \ldots \times X_n$ is the set of inputs of a and $O_a = Y_1 \times Y_2 \times \ldots \times Y_n$ the produced outputs. As an actor is invoked, it returns a set of valid modes from valid elements, expressed as the power set of M_a. This set of "next modes" is important for dataflow designers who want to implement a non-deterministic application which can proceed multiple paths. The modes are checked during the execution of the enabling function and the actor is invoked in the choosen mode. In case of no returned mode

the actor will be disabled. In contrast to PSDF a consumed token is only available in the invoke function itself and not in any subsequent functions [40, 49].

The design principles of EIDF are applied in the implementation of LWDF, which will be further explained in section 3.6 and is limited to a deterministic behaviour. The newly developed LWEIDF implements EDIF without restrictions and will be introduced in section 4.1.

3.2 DSPCAD research work

Computer-aided design (CAD) connected with implementing digital signal processing (DSP) is the research aim of the DSPCAD group which is located at the University of Maryland, USA. A key area of the research are dataflows, not in the sense of dataflow computers, but in a sense of an application modeling and programming methodology for signal processing systems. This scheme is used to analyze and manipulate data streams which can be audio or video streams, images or digital communication waveforms. Processing of data streams is implemented on DSPs, field-programmable gate arrays (FPGAs), multi-core graphic processors and customized integrated circuits. Through the use of formal models and structured programming methods, model-based design can improve productivity of hardware and software system designers and the quality of implementations. Model-based design for signal processing is the central focus of the Maryland DSPCAD Research Group [54].

3.3 Usage of DICE

The package of utilities called DICE (DSPCAD Integrative Command Line Environment) was developed by the research group to support the implementation and testing of software. It is used as a foundation for implementation of the model-based design methodology and offers efficient management for software projects. It supports cross platform operation on Windows, Mac OS or Linux and allows application and integration of different testing methods. The usage of DICE is not limited to DSP operations

but allows a broad usage based on a command line interface to utilize existing tools more efficiently [7, 8].

DICE is a combination of implementations in C, Python and bash scripts. Therefore, tools for compilation have to be existent on the machine which should be equipped with DICE. A detailed description of the installation process can be found in the technical reference [8].

The whole process of developing and testing of implementations for the research of this thesis has been done with the support of DICE. Using the available capabilities had a positive influence throughout the whole process of the research. The following two sections describe the most important usages which are only a small subset of the total range of applications. It includes an effective navigation throughout directories with stacks and a clipboard which is explained in [7].

3.3.1 Compile process

The package specification `dicelang` provides different features which also include a build process. When building software packages with DICE, "dlcconfig" replaces the common "Makefile". This file offers a simplified API which uses a set of customized makefiles. This allows the creation of a library with simple commands noted in 3.1.

```
1 dlcincludepath="-I. -I$UXLIDEC/src/gems/basic -I$UXLIDEC/src/runtime"
2 dlctargetfile="lide_c_rtp.a"
3 dlcobjs="lide_c_rtp_read.o lide_c_rtp_write.o"
```

Listing 3.1: Content of a dlcconfig for building a library

This makes writing of the "dlcconfig" file as easy as the process is abstracted: compiling "dlcobjs" with the "dlcincludepath" as "dlctargetfile". Further options are a special install directory, using existing libraries and a verbose mode. In example 3.1 the two actors read and write, implemented in LWDF, are compiled and supplied as a library.

3.3.2 Testing process

DICE offers a testing framework which is used mainly for unit tests but it can be also used for higher levels of abstractions, including subsystem and system tests. Same as the build process, the testing framework is a flexible and lightweight environment. Different test suits are implemented in separate directories. In each of these directories are subdirectories (referred as individual test subdirectory - ITS) which contain the specific tests. All these specific tests use one program that processes the input data and the ITS name has to start with "test". This allows multiple test suits with different test cases [7].

An individual application for a test suite is stored usually in a folder called "util" and contains the source code for the so-called driver. The compilation process is realized by a similar "dlcconfig" which is used in the library build process (see 3.1). It depends on the suffix of "dlctargetfile" which kind of makefile will be for the build process, e. g. "exe" is creating a executable application. The required library has to be referred in the "dlcincludepath". To run a test suite, the file "runtest" is essential (stored in the "util" folder) which calls the driver with the parameters, e.g. input and output files. It is named by the "runme" file in the ITS which is called during the execution of a test suite [7].

An ITS contains several files which are listed beyond - including a short description [7]:

- `runme`: calls "runtest" file in the "util" folder
- `makeme`: calls "makeme" file in the "util" folder
- `correct-output.txt`: contains the output which is produced by the driver in this certain test case
- `expected-errors.txt`: contains the error-output that is created by the driver in this certain test case
- `input-files`: can be different file types that contain the input data for a certain test case (if required). This information has to be specified in the "runtest" file which is found in the "util" folder.

- `README.txt`: contains a short test description

All test suits can be started with the command "dxtest" which has to be executed in the parent directory of the different test suits. This command goes recursively through all subdirectories, runs the "makeme" and "runme" files in every test folder and returns a report summary. The example in listing 3.2 has five test suits with one ITS each, one fails due to different output than in `correct-output.txt`, the other for the reason of another error output then in `expected-errors.txt`.

```
1 All tests are complete.
2 Summary of test results:
3 Test count: 5; test failures: 2; error output mismatches: 1
```

Listing 3.2: Report generated by the dxtest command

This report allows a fast verification of several usages of included libraries. It helps to discover side effects and logical errors in the code. Especially for the implementation of actors in a dataflow, DICE unit testing allows simple and effective test routines.

3.4 Dataflow Interchange Format

DIF was developed in an academic research to specify and manipulate dataflow models like the SDF (see 3.1.1). In further versions other dataflow models were supported and more detailed dataflow semantics like actor specifications were able to be described. The DIF language has a syntax which is based on the dataflow theory and is not dependent on any design tool. Therefore, different types of dataflow applications can be described in DIF including the dataflow graph by listing nodes (actors) and edges. In a block called "actor" specifications on the parameters and modes of the actor can be made. DIF is not used to create executable code but to describe dataflow applications. The Code for actual signal processing is implemented in actors. Tools, like the DIF-to-C compiler, are able to create a structure where actor code can be implemented. In listing 3.3 an example can be found which describes the dataflow in figure 3.1:

```
1  sdf sdfDemo1 {
2   topology {
3    nodes = A, B, C;
4    edges = e1(A,B), e2(B,C);
5   }
6   production {
7    e1 = 1; e2 = 1;
8   }
9   consumption {
10   e1 = 1; e2 = 1;
11  }
12 }
```

Listing 3.3: Example of a dataflow description with DIF

Further reading on DIF, its structure, availible blocks and more can be done in [19, 20, 41].

3.5 Targeted Dataflow Interchange Format

Targeted Dataflow Interchange Format (TDIF) extends the DIF format by dynamic dataflows as described in EIDF (see 3.1.3), cross platform actor support and dataflow design. TDIF uses a subset of EIDF called core functional dataflow (CFDF) which allows both static and dynamic behaviour but supports only deterministic implementations. More details on CFDF will be explained in the next section that deals with LWDF (see 3.6). The TDIF language as a high level specification format is used to specify dataflow actors and define the connection between them. The lightweight implementation for actors contains five keywords: *module*, *input*, *output*, *param* and *mode*. Type and name are defined in the *module* and will be used throughout the dataflow (see listing 3.4 - line 1). The keywords *input* and *output* define ports of the actor which will be connected to edges. *Param* defines possible parameters which can be set during the actors initialization and *mode* can give multiple modes for the actors operation. An example of a full actor definition is given in listing 3.4 [52].

```
1 module C rtp_packetizer
2 input input1 short
3 input input2 int
4 output output1 short
5 param size int
6 mode init
7 mode process
8 mode send
```

Listing 3.4: Example of an actor description with TDIF

The TDIF language can be interpreted by different compilers, which create an application programming interface (API) for the dataflow implementation of the actor. Currently two compilers are available: TDIF-to-C and TDIF-to-CUDA for DSP's of Texas Instruments. The result of the compilation process in C are header files: "[actor-name]-auto.h" and "[actor-name].h". The generated interfaces in those header files can be used by the programmer of actors to build his implementations. Three important parts have to be developed or defined: the invoke function, which does the actual processing of the input data and the consumption resp. production rate of the actors in the different modes [52].

Relevant runtime information is encapsulated in data structures which is referred to as *context*. Correspondent data for actors is stored in the execution context (EC) which includes parameters and state variables. The topological context (TC) encapsulates port information which define how actors are connected to other parts of the dataflow. Actor implementations have access to both of the data structures which are included in the runtime library [52].

In order to run a dataflow a scheduling algorithm has to be implemented. It has an important impact on performance and memory usage. The driver of the dataflow initiates actors and FIFOs and calls a scheduling algorithm. This can be a canonical scheduling or any other customized scheduling algorithm. It always has to be implemented manually but the driver can be created by the TDIF compiler. An example has been already given in listing 3.3, the keyword "sdf" has to be replaced by "cfdf". As a lightweight language, TDIF has limited capabilities but the compiled driver can

always be extended [52].

3.6 Lightweight Dataflow

The lightweight implementation of EIDF is called LWDF. In order to fulfill the restrictions of the DIF language, CFDF as a subset of EIDF is used. The main difference is the modified equation of the invoke function which is returning only one possible "next mode" instead of a set (see equation 3.4).

$$\iota_a : (I_a \times M_a) \rightarrow (O_a \times M_a) \tag{3.4}$$

Advantages of the LWDF are the minimal dependencies on libraries and high flexibility in the design process. Both static and dynamic dataflows can be implemented and a lot of different applications have already been realized in the last years [51].

3.6.1 Actor design

The three interface functions of an actor in LWDF reflect the lifecycle: *construct*, *execute* and *terminate*. The construct function can be extended with initiating parameters which are necessary for the actors operation. In the function itself the actor connects to FIFOs and performs actor related initiations like opening a filehandler. Closing filehandlers is performed in the terminate function together with other clean-up sequences. The execute functions are split in enable and invoke. In the enable function the actor is checked whether it is able to be invoked. Actual firing of the token is executed in the invoke function. To describe the functionality of LWDF the following definitions are made [49]:

- inputs(a) / outputs(a): a set of input / output edges of actor a
- population(e): the number of token in a FIFO e before the execution of an actor
- capacity(e): the size of FIFO e
- cons(a, m, e) / prod(a, m, e): rate in which an actor a consumes / produces tokens in mode m from / to a FIFO e.

These definitions are used to calculate the return value of the enable function which can be either true or false. This is explained in more detail in chapter 4 which deals with modelling and implementation.

3.6.2 FIFO design

LWDF implements FIFO operations which are encapsulated in interface functions in C. They can be called from a driver and from an actor. They include

- creation and destruction of a FIFO,
- read and write operations to a FIFO and
- getting the population and capacity of a FIFOs.

As defined in section 3.6.1 actors are dependent on FIFO functions. Tokens can be single values like integer or double and also pointers to larger data structs. This allows flexibility in storing different kind of data. Every FIFO has a defined maximum size. Behind the interface of the FIFO any kind of other implementation can be done [49].

3.7 Scheduling Algorithms

A valid scheduling algorithm is deadlock free and has a consistent rate of produced and consumed tokens. This means that every token which is produced by an actor is consumed by an another actor. When choosing the right buffer size, it should not lead to an overflow of the FIFO buffer [19].

The most simple algorithm is the canonical scheduler. It executes actors in a defined sequence. In case of EIDF it calls the enable function first and in case of the return value “true” the actor is invoked. The scheduler enables every actor as long as one of the functions return “true”. Scheduling is completed as soon as all enable functions return “false”.

There is a lot of potential in customized scheduling in terms of performance and memory usage. The scheduler can, in case of EIDF, get the mode in which an actor has been

invoked, and adapt the order of the execution or save time due to not enabling actors at all. In the next chapter examples of optimized scheduling algorithms are shown [49].

4

Modelling and Implementation

The functionality of DSP applications can be very well described in dataflows (see [6, 48]. These applications like digital communication or image processing have to be capable of real-time processing. RTP applications have similar characteristics and requirements: meeting real-time constrains is critical and voice packages represent a continues dataflow. ITU-T recommendations (see [22, 23]) describe the algorithms PLC, VAD and CNG as block diagrams, which can be translated from dataflows under certain conditions [35]. Therefore, modelling of RTP applications with dataflows is a promising research. This chapter describes adaptions which have to be made on dataflow models and how RTP algorithms can be modeled in dataflow graphs. It also shows which kind of improvements can be made by modelling RTP applications with dataflows.

A goal of this research was to model and implement an RTP application as shown in figure 4.1. The connection between sender and receiver is bidirectional and contains the same elements in both channels. VAD is applied on the senders side and the receiver deploys the algorithm for PLC first, followed by CNG. As described in section 2.3 and 2.4 quality can be enhanced and network load can be reduced by applying the mentioned algorithms. These improvements have already been used widely in the industry but have not been modelled and implemented as dataflow graphs yet.

The mentioned RTP algorithms will not be implemented as one actor but as a subsystem. Each block in figure 4.1 has multiple actors connected to each other by edges in order to accomplish the task of the algorithm. This segmentation will make it easier

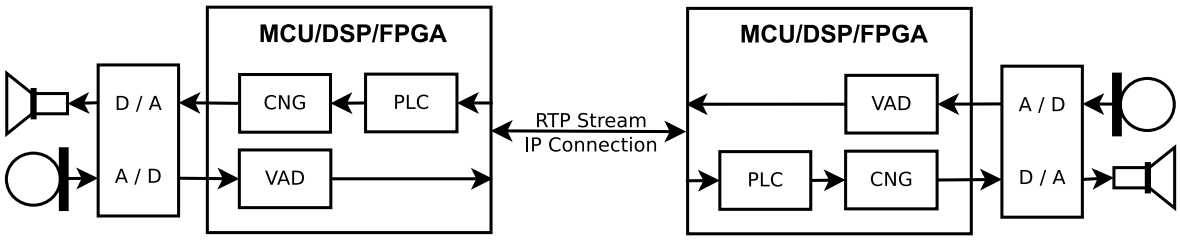


Figure 4.1: RTP Stream between sender and receiver

to develop, test the algorithm and perform rapid prototyping.

Elements like a jitter buffer, RTP session management or any other functionality that is required in real world networks, is absent intentionally and is not a part of the research. Dataflow models are simplified for the reason of less complexity and easier analysis. Nevertheless, the algorithms itself are implemented to meet ITU-T recommendations as a prove of concept.

4.1 The new LWEIDF

Developments of the programming model TDIF (see 3.5) are based on the DIF language. Designers are able use this standard language to specify dataflow models for streaming-related applications. It has been designed to provide a general framework to represent dataflow graphs in a variety of applications. Specifications and formalism of the DIF language allow only deterministic behaviour. Therefore, the programming model TDIF supports deterministic actors with a single “next mode” [20].

Even though the programming model LWDF (see 3.6) is derived from the non-deterministic EIDF (see 3.1.3), it has been developed with similar restrictions to meet the DIF formalism [41]. This special case is referred to as CFDF.

When modelling RTP applications designers have to deal with unpredictable events and states. This can be a sudden packet loss (see 2.3) or a detection of voice in a noisy stream (see 2.4). To handle this in a deterministic environment, the actors would have to send dummy tokens to simulate an inactive state under certain conditions like a packet loss.

To support non-deterministic behaviour a new model is introduced: LWEIDF. As the model from which it is derived from, the invoke function can return a set of valid modes. The enable function matches available tokens in the input and output FIFOs with requirements of the assigned modes and sets a mode in which the actor is invoked. Until the enable function is called (this happens usually right before the execution of the invoke function) actors are able to operate in one of the valid modes. Therefore, the new LWEIDF can be used to model non-deterministic applications. The invoke function is still permitted to return only one mode. For this reason it is still possible to model deterministic dataflows.

The following sections will show the effect of non-deterministic dataflows to RTP applications and will explain the functionality and implementation of the new model.

4.1.1 Deterministic vs. Non-Deterministic

In the process of modelling RTP algorithms with dataflows the question raised, how to deal with absent packages. The usage of dummy packets was evaluated in the existing programming models LWDF and TDIF for RTP applications: a workaround for VAD resp. CNG algorithms is a transmission of a packet with an empty payload instead of the 160 samples. This would still save bandwidth but would soften the concept of the different modes. The same mode would have to be invoked in case of a voice and a noise signal. After receiving a packet as a token, the content has to be determined in order to identify voice or noise. On the basis of this decision conditional blocks have to be called. Each actor which is a part of the dataflow has to have certain blocks in a main mode. The more actors are involved, the higher is the overhead produced by dummy tokens. It is not possible to send a packet with specific information about the coming noise period. Needed information like time span is not available, because RTP applications in connection with VoIP need to be processed in real time. So there is no possibility to switch an actor back into a voice mode after a noise period.

Using a non-deterministic model, an actor can be invoked in a set of modes. This means in the case of VAD resp. CNG that the VAD subsystem will not produce any packets in the case of noise. The CNG subsystem will not get a packet but will be able

to switch in another valid mode to create a packet with synthetic noise. In case of noise a temporary sink and source is emerged instead of an edge with dummy actors. Using a non-deterministic dataflow makes it easier and clearer to design and implement RTP applications.

In case of a packet loss (see 2.3) no predictions can be made. When no packet is transmitted the PLC algorithm has to create a package. Simulation can be done when using dummy packets but this will lead to massive drawbacks. In this case a non-deterministic dataflow is the enabler to model a design which is close to reality.

In a deterministic dataflow model each mode of an actor has a constant consumption and production rate [49]. This is not true for a set of modes but after the execution of the enable function and a selection of a mode consumption and production rate for executing the invoke function are constant.

4.1.2 Next mode and truth table

When the invoke function is executed, it sets a "next mode". Both LWDF and TDIF use this kind of implementation (see [49, 52]). EIDF offers that a set of modes is returned. This will give multiple modes a chance to be invoked. Table 4.1 shows an example of the functionality of the new model.

		Input			Output
Mode	Valid	A	B	C	A
A	1	0	1	1	1
B	0	1	1	0	1
C	0	0	1	1	1
D	1	1	1	0	1

Table 4.1: Available modes in an example actor of LWEIDF

The shown example of an LWEIDF actor offers four possible modes. As in other models the number of modes is free to choose for designers resp. developers. At the end of the invoke function a mode for the next execution is set:

```
1 next_mode = A or D
```

Listing 4.1: Abstract definition of `next_mode`

This declares mode A and D as valid for the next execution. Input and output values in the table are boolean, based on the rules of the enable function of LWDF (see 3.6 and [49]):

$$population(e) \geq cons(a, m, e) \ for\ all\ e \in inputs(a) \tag{4.1}$$

$$population(e) \leq (capacity(e) - prod(a, m, e)) \ for\ all\ e \in outputs(a) \tag{4.2}$$

Thus the enable function checks return values of inputs as described in equation 4.1 and outputs as in equation 4.2. As soon as the values match the given table 4.1, a single mode is set [49]. For mode A this would mean:

- not enough tokens are available on input A
- enough tokens are available on input B and C
- a token can be produced on output A

In this case mode A would be set and the enable function returns “true”. If the constellation is different, mode D is checked and can be set if all inputs match. Otherwise the enable function returns “false” and the actor is not invoked. Mode B and C do not need to be checked because they are not present in the set of valid modes. The scheduler calls the enable function again after the previous actor has been invoked. The set of valid modes stays the same, even if the enable function returns “false” multiple times.

4.1.3 Bit Patterns and Macros

In LWDF (see 3.6) and TDIF (see 3.5) each mode is associated with an integer value. For a better understanding and management of the source code a macro is defined as a placeholder (listing 4.2 - see [49]):

```
1 #define LIDE_C_RTP_PLC_MODE_WRITE          1
2 #define LIDE_C_RTP_PLC_MODE_CREATE         2
3 #define LIDE_C_RTP_PLC_MODE_INACTIVE       3
```

Listing 4.2: Macro definition for actor modes

To maintain compatibility to LWDF a set of "next modes" is not rendered in an array of integer values. The n-bit of an integer value is set if the n^{th} mode of an actor belongs to a set of valid "next modes". In the case of the definition of MODE_WRITE and MODE_INACTIVE bits would look like this (listing 4.3):

```
1 bit 8  5 4  1
2    |0000|0101| = 5
```

Listing 4.3: Bits and value for the set of valid `next_modes`

To make the handling clearer and easier to manage the following macros are introduced (listing 4.4):

```
1 #define SET_MODE(mode)          (1<<(mode))
2 #define CHECK_MODE(v,mode)    (((v) & (1<<(mode))) & (SET_MODE(mode)))
```

Listing 4.4: Macros for `next_modes` bit operations

The SET_MODE macro performs a simple bit shift. Developers are able to define a set of valid by using this macro and an inclusive OR operator (listing 4.5):

```
1 next_mode = SET_MODE(A) | SET_MODE(B)
```

Listing 4.5: Usage of the SET_MODE macro

The CHECK_MODE macro (listing 4.4 - line 2) is used in the enable function of actors. It checks if a given mode is part of a set of valid modes. Variable v is an integer value with a set of modes encoded in bits as shown in listing 4.3. If an actor is not part of a set of valid modes, "0" is returned. If a mode matches a set of valid modes, the number of this mode is returned.

In case of the example actor in table 4.1 several inputs and outputs have to be checked with the boolean functions 4.1 and 4.2. To simplify this verification four other macros are introduced in LWEIDF (listing 4.6):

```
1 #define INP(i)      (lide_c_FIFO_population(i)>0)
2 #define OUT(i)      (lide_c_FIFO_population(i)<lide_c_FIFO_capacity(i))
3 #define INPM(i,v)   (lide_c_FIFO_population(i)>=v)
4 #define OUTM(i,v)   (lide_c_FIFO_population(i)<=(lide_c_FIFO_capacity(i)
      -v))
```

Listing 4.6: Macros for checking on equation 4.1 and 4.2

The macros for input and output in line 3 and 4 map exactly the boolean functions as defined in equation 4.1 and 4.2. Variable i is a number of inputs and v is a consumption rate (line 3). The OUTM macro works equivalent: i is a number of outputs and v is a production rate. A "0" is returned if not enough tokens are available in the FIFO-buffer or if there is not enough space in the output buffer. Otherwise the return value is "1". The macros in line 1 and 2 are shortcuts for the two other macros. Using INP() checks if at least one token is available, OUT() check for the FIFO space for one token.

This approach limits the number of possible modes to 31 resp. 63 in 64-bit systems. However, the number of modes per actor will usually not exceed ten modes even in complex dataflow designs.

4.2 Design Phase

Dataflow design is about describing algorithms in terms of important components for their functionality. It describes a stream orientated flow of data between modules. Designers deal with signal flow block diagrams and identify repetitive computations. During a design process feedback loops have to be identified, possible parallelisms and multidimensional computations have to be discovered. However, models should not be over-specified, defined is only what is needed for correctness. This includes constrains and logical descriptions. One signal flow block stands for a specific processing of signals. This can be an inversion or even a more complex computation. A path in a diagram shows different processing steps of a signal during one flow [9].

As described earlier, the RTP algorithms are build as subsystems. They consist of

multiple actors and edges. Each single actor can be tested first, followed by a subsystem and a combination of subsystems as shown in figure 4.1. Individual actors can be replaced to change the behavior of a whole system. This can be used for rapid prototyping and will be further explained at the end of this chapter.

The concept of EIDF has been used to model RTP applications because it allows dynamic behaviour and the concept of mode switches fits very well. Furthermore, API's are availible for an implementation and audio applications have been already modelled and implemented [51].

4.2.1 Simplify the model

For development, testing and evaluation the real world environment has to be simplified. To reveal this process of modelling and simulation, intermediate steps are explained on basis of the PLC algorithm. Figure 4.2 shows three major steps.

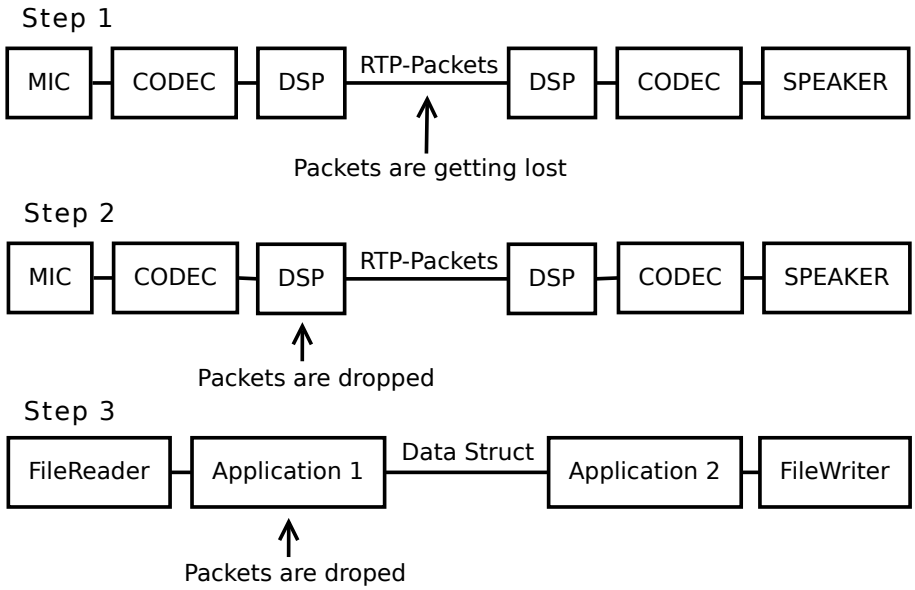


Figure 4.2: Modelling steps from a real world example

As the first phase, a model should represent an application how it appears in a real world surrounding. Through routing problems and system overload IP packets get lost. This happens unpredictably and is strongly connected to the external circumstances and to the distance between sender and receiver. Using this environment it is not

possible to assemble reliable tests and comparisons. The second step is already meant to be performed under laboratory conditions. Reasons for lost packages are eliminated and stochastic algorithms drop packages. Thereby certain conditions can be simulated and test cases can be repeated. Propagation delay is reduced and it is easier to focus on processing time. As a last step microphone and speaker are replaced by defined sound files to make results reproductible. Thus waveforms can be compared and several test cases can be executed. Only one hardware is needed and simple data structs are exchanged. The propagation delay is fully eliminated. Remodelling dataflows and test cases is possible in a very flexible way. This will not make real world tests unnecessary but major testing can be done in a far cheaper laboratory environment. It helps to discover a large percentage of mistakes and makes the final product more reliable. The process of simulating and testing is an important part in dataflow development and will be explained in this chapter.

4.2.2 Modelling simple PLC

As explained in section 2.3.1 a simple repeat of a packet during a time of packet loss can increase the quality of an audio stream. To perform this task one actor is sufficient. The two necessary modes are shown in table 4.2.

	Input	Output	
Mode	A	A	Comment
forward	1	1	stores and forwards the packet
fill	0	1	copy and send the stored packet

Table 4.2: Single actor modes table for simple PLC

LWEIDF has to be used to implement this actor. In both modes the actor has to be able to switch to another mode right before the execution when an input token is possibly available. In “forward” mode, a packet is not modified and the actor behaves like a FIFO. In the “fill” mode it operates like a source actor and produces a token on every execution. To switch to this mode, the `drop-packets` actor has to be placed before the `simple-plc` actor. This actor will be explained in section 4.3.4. Implementation,

simulation and testing of this single actor is a first step on the way of working with larger subsystems.

4.2.3 Modelling PLC

The original implementation of the algorithm in [23] is using a global buffer for signal history. It is used to store the signal of the last 48,75 ms (see 2.3.2). When splitting the algorithm in multiple actors three different possibilities come up, how to handle this history buffer:

- The buffer is existing multiple times
- The buffer pointer is spread during initialisation of actors
- Different buffers are designed according to the needs of the actors

One problem with the first point is, that buffers of the different actors have to be in sync. To achieve this, a lot of FIFOs between the actors and administration effort is needed. In addition it requires an application as a whole to allocate more memory then it really needs. On the other hand, using a pointer which is used by every actor in the subsystem breaks with the idea of individual actors. It should be possible to test individual actors without initializing other actors which allocate memory to make these actors usable. Therefore, a new buffer design has to be developed which meets the needs of every actor.

When analyzing the function description [23] of the PLC algorithm three major blocks can be found. The first one is an history management block: it writes good voice samples into a buffer and retrieves them if a packet is lost. The second one is the algorithm itself, which searches the signal for a pitch and creates a new synthetic signal. The third block is a mechanism which handles transition between original and generated signal. The subsystem is designed according to this segmentation and can be seen in figure 4.3. Its design is general, therefore it can also be used for modelling of simple PLC (see 4.2.2). Actor `gen_syn_g711` is replaced with a simpler `gen_syn_simple` actor which only uses one last packet to create a new packet.

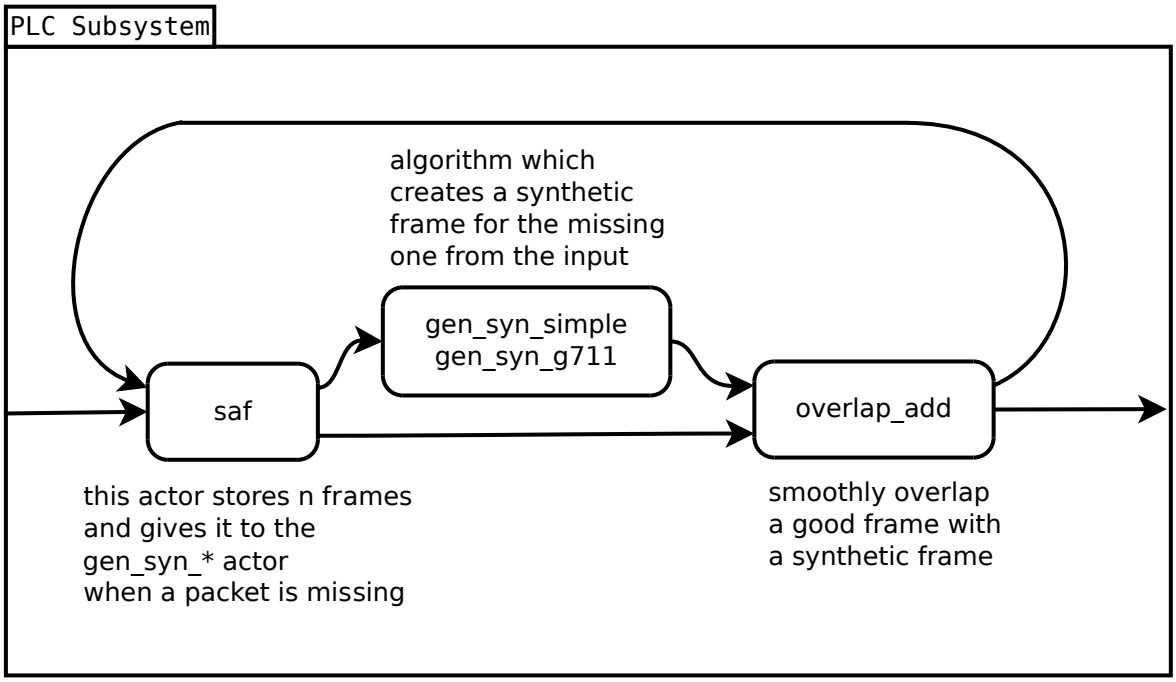


Figure 4.3: Dataflow model of PLC G.711

Even other PLC algorithms (see 4.4.5) can be easily modelled into this subsystem. Most of the algorithms which are based on creating synthetic frames of the signal history contain a history buffer and an overlap functionality. This makes it possible to test and evaluate new PLC algorithms with less effort and expense.

More details concerning the individual structure of the involved actors are given in section 4.4. An overview of the usage of this subsystem can be found in appendix A.

4.2.4 Modelling VAD

The VAD algorithm as implemented in [22] is well suited for dataflow modelling. It consists of multiple calculations which can be done in a row or even parallel. As described in section 2.4.1, the algorithm preprocesses the signal first and performs several calculations. Core of the algorithm is a decision if a packet is voice or noise. In addition to this, noise parameters have to be handled and updated if changes happen. Dataflow modelling for this algorithms can be found in figure 4.4.

This dataflow is designed to possibly run in a multi-core environment. The DSPCAD

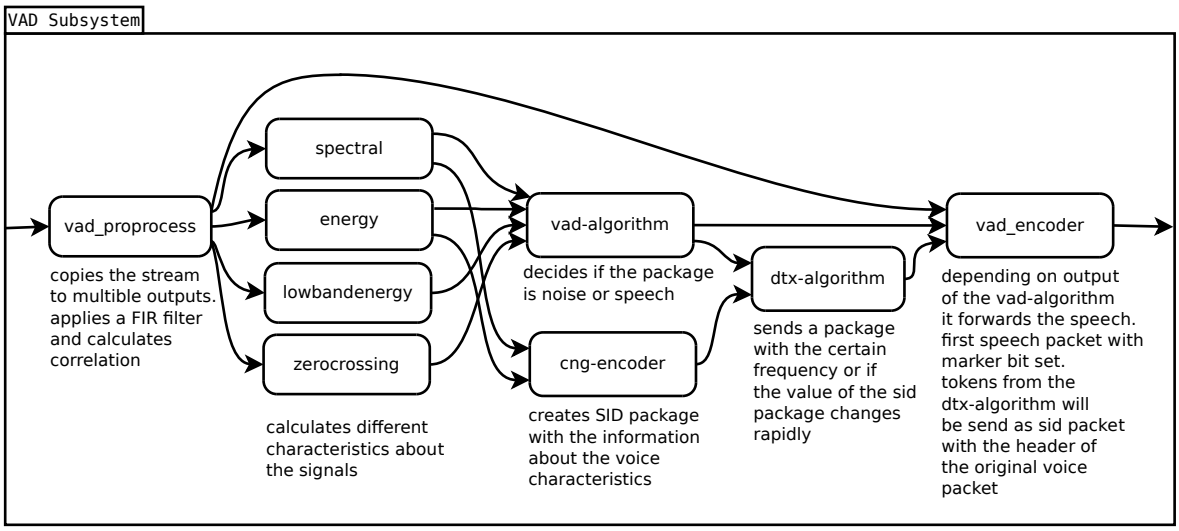


Figure 4.4: Dataflow model of VAD G.729

research group has already done research on this kind of applications in the past (see [12, 11, 16, 10]). Expensive operations like calculating signal characteristics can be performed parallel.

An RTP packet has only one path where it is transferred through the subsystem. It is marked by the actors `vad_preprocess` and `vad_encoder`. All other FIFOs will only transfer tokens of integer, float and pointers to float values which contain audio samples. The last actor decides if a RTP packet is send or not.

Individual structure of involved actors with more details are given in section 4.5. Usage of this subsystem can be found in appendix A.

4.2.5 Modelling CNG

Inputs of the CNG subsystem are packets with voice samples and information about noise characteristics in the period of noise. They have to be split and CNG packets have to be processed. This task is performed by the CNG decoder actor. This actor can contain different implementations of decoding SID frames. Encapsulation makes it possible to take over the rest of the subsystem without variations. A `mux` actor will create a stream of packets which will be similar to the VAD input. The subsystem model is illustrated in figure 4.5.

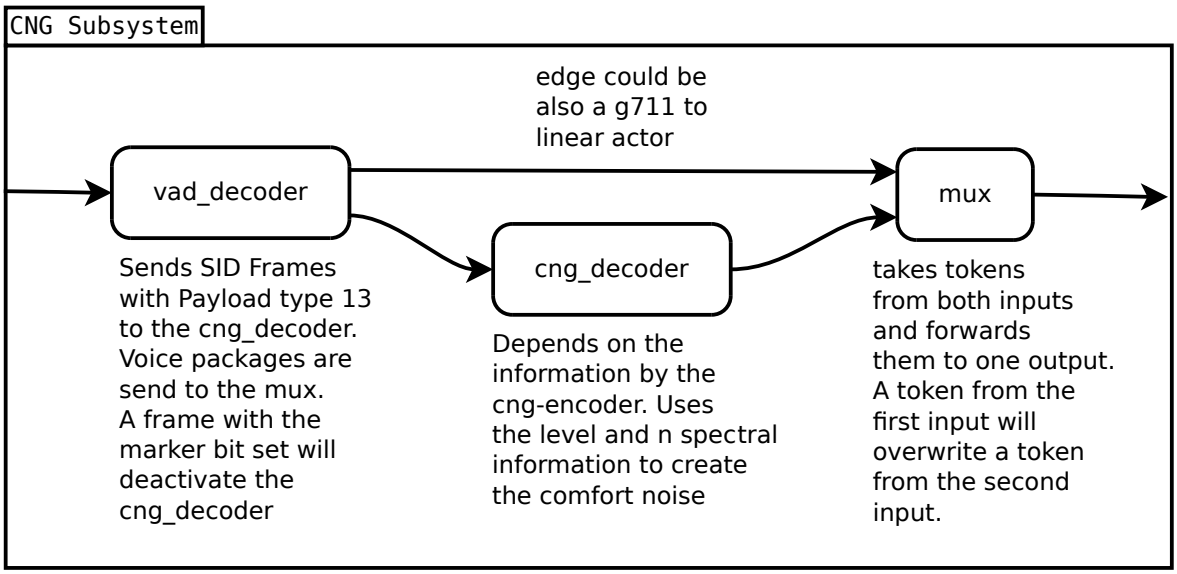


Figure 4.5: Dataflow model of CNG G.729

Whenever voice packages have another format then the output of a CNG decoder, a conversion actor has to be used. This could be the case if a decoder produces 16-bit linear samples and voice packages are G.711 encoded. Then a conversion actor `g711-2-linear` (see 4.3.3) has to be placed between the `vad_decoder` actor and a `mux` actor. It would also be possible to use a `linear-2-g711` actor and plug it between the `cng_decoder` actor and the `mux` actor. It is important that a `mux` actor receives equal encoded audio samples at both inputs.

More details about the individual structure of involved actors are given in section 4.6. Appendix A provides an overview of usages of this subsystem.

4.2.6 Modelling other algorithms

Algorithms are rarely defined as dataflows. The representation is often done in block or flow diagrams. Based on modelling in this thesis a general procedure can be given. The first step is a definition of functional blocks which can operate individually. It is important to generalize functionality of actors in order to allow re-use. Based on the desired behavior actors have to be defined if they should operate deterministic or non-deterministic. The last step is to describe the flow of tokens between actors.

4.3 Basic components

Newly developed LWEIDF inherits all features from the existing LWDF. This includes FIFOs, common actors and utilities. Additional features have been described in section 4.1. Using subsystems to model RTP applications requires some additional actors e.g. for codec conversion or simulating packet loss. A RTP struct is defined to pass on packets between the different actors. In addition, a real time scheduler will be introduced in this section.

4.3.1 Fifos and Datatypes

A standard FIFO in LWDF in its C-based implementation can perform operations which are encapsulated by interface functions. Due to the usage of polymorphism, different implementations can be used through the same interface. This makes it possible to focus on a design first and develop specified FIFO implementations later. A FIFO can store arbitrary data types. Developers are free to use integer, float and character datatypes or pointers to any kind of data. The interface offers creation of new FIFOs, read and write operations and a check of capacity or current number of tokens.

When modelling RTP applications a token often refers to a pointer which refers to a RTP packet. Although transfer of packages is a main task of an application, the VAD subsystem transfers many integer values and pointers to blocks of samples. This flexibility of edges helps to use various data types to design very efficient dataflows.

As explained in section 2.1, RTP packets themselves are wrapped into other packets based on rules of the OSI model [53]. During the processing of RTP packets, data of wrapping packets are not essential for an application. Therefore, only RTP headers and the appendant payload are transferred through the dataflow and used subsystems. A RTP packet datatype consists of header and payload. To simplify the handling of a payload, the struct is extended with a `plength` value (listing 4.7 - line 14). Usually a payload length is defined by a payload type. The recommended structure by the ITU-T (see [47]) is used for the header (listing 4.7 - line 1).

```
1  typedef struct {
2          unsigned int version:2;   /* protocol version */
3          unsigned int p:1;         /* padding flag */
4          unsigned int x:1;         /* header extension flag */
5          unsigned int cc:4;        /* CSRC count */
6          unsigned int m:1;         /* marker bit */
7          unsigned int pt:7;        /* payload type */
8          unsigned int seq:16;      /* sequence number */
9          unsigned int ts;          /* timestamp */
10         unsigned int ssrc;        /* synchronization source */
11         unsigned int csrc[1];     /* optional CSRC list */
12 } rtp_hdr_t;
13
14 typedef struct {
15   rtp_hdr_t header;       /* full header */
16   unsigned int plength;   /* length of the payload */
17   void * payload;         /* pointer to the payload */
18 } rtp_packet;
```

Listing 4.7: Datatype for RTP Packet

The allocation of memory for the structure `rtp_packet` in actors is done in two steps. First packet structure and header is allocated (listing 4.8 - line 1) followed by the payload (line 2).

```
1 rtp_packet * packet = malloc(sizeof(rtp_packet));
2 packet->payload = malloc(bytes_per_sample*samples_per_frame);
```

Listing 4.8: Allocating memory for an RTP Packet

Values for `bytes_per_sample` and `samples_per_frame` have to be available in every actor which produces RTP packages (see 4.3).

4.3.2 Read and write Actors

A sample implementation of TDIF actors already contains source code for reading and writing audio files (see [54] - TDIF package). This code has been used to read and

write the header of wave files, which provide data for unit tests of RTP applications.

The `read` actor starts in the "init" mode, reads the header of a file and sets "write" mode as "next mode". A file is read until the end of file (EOF) and produces RTP packets with a given payload length. It is defined by

$$bytes\_per\_sample * samples\_per\_frame = 2 * 80 = 160 \tag{4.3}$$

for a packet with a length of 10 ms and 16 bit linear audio samples at 8 kHz sampling rate. Pointers to packets are send through the `out` edge. After EOF is reached, the actor sets "wrapup" as "next mode". This outputs the value "1" at the `out_terminate` edge. Using this behaviour, the file size does not have to be read in the beginning (like in the TDIF actors) to determine the length of an audio file. A file reader could be replaced by a socket to read a stream whose length is unknown. As soon as the stream is closed, the actor switches to "wrapup" mode.

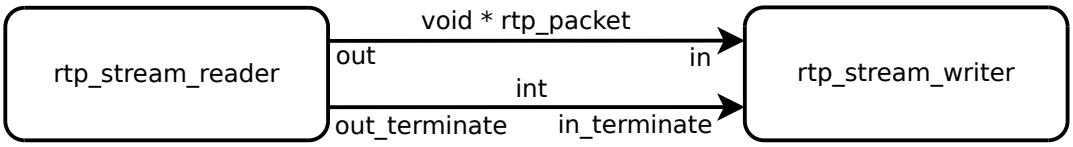


Figure 4.6: Read and write actors connected through two edges

The `write` actor is designed equivalent. After writing a header in "init" mode, `input_in` is receiving pointers to RTP packages and `in_terminate` is used to put the actor in "wrapup" mode. To achieve this non-deterministic behaviour, LWEIDF is applied. The actor can be invoked in four different modes, based on inputs and sequence of inputs. A mode table is presented in 4.3.

In "write" or "zeros" mode, possible "next modes" are

- "write": for another incoming packet which payload is written in the wav file
- "zeros": writes silence if no incoming packets are available, but the in_terminate input has not received the terminate signal
- "wrapup": to update the wave file header with the current stream size counter and close the file handler

	Input		
Mode	out	out_terminate	Comment
init	0	0	writing header at first execution
write	1	0	write payload from RTP packets
zeros	0	0	writing zeros for missing RTP packets
wrapup	x	1	updating file size and close

Table 4.3: Available modes in the LWEIDF `write` actor

An initiation of actors can be found in listing 4.9. In addition to the file handler and FIFOs more parameters are needed to set up an actor: `bytes_per_sample` and `samples_per_frame` determine the length of the payload (see equation 4.3). The read actor needs two more values, which have to be set by the driver: timestamp and sequence number. As described in 2.1.1 they have to be set randomly.

Creating a dataflow with just those two actors like in figure 4.6, results in equal file content of input and output file. Source code of this actors is available in appendix B. A driver for the dataflow is existing in the folder demo/transmission.

4.3.3 Conversion Actors

Conversion from a linear encoding to a G.711 or G.729 encoding is a standard operation and described in different recommendations by ITU-T ([23, 22]). To use this functionality in applications with LWEIDF two actors are created:

- `g711_encode` with encapsulated linear to non-linear conversion based on a reference table
- `g711_decode` as an inverse function to the above

No special initialisation is needed for both actors. A single input and output is connected to an edge.

A conversion from a non-linear to a linear encoded signal makes following calculations faster and easier to handle. If needed, transformation in an original encoding has to be performed in the end of a subsystem resp. of a dataflow before the sink (`write`) actor. Besides the source code for the actors a demo test is filed in demo/transmission_g711 (appendix B).

4.3.4 "Drop Packages" Actor

In order to test a PLC actor and a PLC subsystem an instance has to be created which simulates packet loss. As described in 2.3 a loss happens unpredictable, therefore an actor would have to drop packets randomly. On the other side, analysis of audio samples which are concealed randomly can make the process of development and testing more difficult. However, if an actor is implemented, it has to be initialized with an inverted drop rate. To cover both cases an actor is designed which is able to operate in four modes, all of the modes expect to receive an input token:

- "write": This is one possible mode in which the actor is initialized. Packets are simply forwarded and a counter is incremented on every execution. When the counter matches the drop rate "next mode" is set to "drop".
- "drop": The actor reads an input token, frees the memory of the payload and packet and sets "next mode" to "write".
- "random_write": This is another possible mode in which the actor can be initialized. A random in the range of "0" to n (n . . . drop rate) rate is generated. When a number matches the drop rate "next mode" will be set to "drop".
- "random_drop": The mode also creates a random number in the same range as "random_write". When a number matches the drop rate again, the actor stays in the current mode, otherwise it switches back to "random_write". This implementation makes it possible that two packages in a row will get dropped.

All of the modes have constant consumption and production rate. The higher an inverted drop rate is set, the less packets will get lost. To achieve a drop behaviour which is closer to reality, algorithms have to be used which are mentioned in section 4.3.4. Source code of the actor only covers modes "write" and "drop", "random_write" and "random_drop" mode are not implemented in the attached source code in appendix B.

4.3.5 Simple PLC Actor

The development of this actor was a first step to test the capabilities of the new LWEIDF. Without using the PLC subsystem this actor performs a simple PLC (see 2.3). Only three other actors are needed to get results. The dataflow is pictured in figure 4.7.

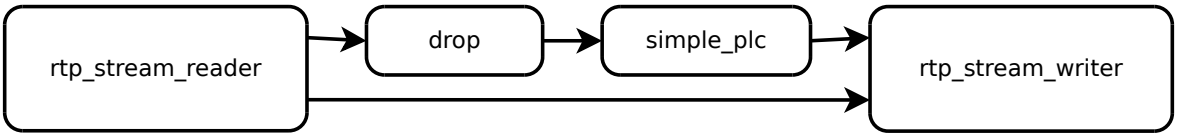


Figure 4.7: Simple PLC dataflow

A `simple_plc` actor has an internal buffer which is always updated with the latest received packet. A packet loss is concealed with this buffered packet. An internal counter makes sure, that a packet is not repeated more than three times. If more than three packets are lost, silence is created as an output packet.

The structure of this driver is used to execute unit tests and will described in the next section. Source code and drivers for this dataflow including possible applications can be found in appendix B.

4.3.6 Drivers for unit testing

Actors are connected by edges resp. FIFOs to a program referenced as driver. In the C-based implementation of LWDF / LWEIDF this driver consists of the following major parts (each bullet will refer to a line in listing 4.9):

- inclusion of header files of actors and FIFOs to have the definition of Abstract Data Types available (ADT - line 1, 2)
- initiate actor and FIFO ADTs (line 3, 4)
- creation of buffers (line 4)
- creation of actors and connecting them by buffers (line 5-10)
- execution of a dataflow graph (line 11)

```
1  #include "lide_c_FIFO.h"
2  #include "lide_c_rtp_read.h"
3  lide_c_actor_context_type *actors[ACTOR_COUNT];
4  lide_c_FIFO_pointer FIFO = lide_c_FIFO_new(BUFFER_SIZE, token_size);
5  actors[ACTOR_RTPSOURCE] = (lide_c_actor_context_type
6        *)(lide_c_rtp_read_new(in_file, FIFO, bytes_per_sample,
7        samples_per_frame, timestamp, sequence_number));
8  actors[ACTOR_RTPSINK] = (lide_c_actor_context_type *)
9        (lide_c_rtp_write_new(out_file, FIFO, bytes_per_sample,
10       samples_per_frame));
11 lide_c_util_simple_scheduler(actors, ACTOR_COUNT, descriptors);
```

Listing 4.9: Simple structure of a driver (not executable)

A driver which connects multiple actors can be complex but will always follow the same structure. A scheduler function enables and executes the actors until the return value of the execution of the enabling function of all actor returns “false”. This method is called canonical scheduling (see section 3.7). It can have a positive influence on the performance of a dataflow if a tailored scheduling algorithm is used. Another reason for a development of a scheduling algorithm is given in the next section.

4.3.7 Real-time scheduler

Using a canonical scheduler, actors will be called repeatedly until the dataflow has processed all data. Therefore, a RTP Stream of 14 seconds length will be transmitted in less then a second [1]. This is useful for testing RTP applications with sound files but not for processing of real time RTP streams. An RTP packet arrives only every 10ms and because of an absence of more packets the dataflow will quit after processing one packet.

To solve this problem a POSIX timer in Linux is used to execute actors once every 10ms. The `timer_set` function can only transfer one pointer to the `on_timer` function. To transfer all required information an ADT is created. It holds a status flag and pointers

[1] using a Intel(R) Xeon(TM) CPU 3.00GHz

to actors including a description (see listing 4.10).

```
1     data->run = TRUE;
2     data->actors = actors;
3     data->actor_count = actor_count;
4     data->descriptors = descriptors;
5     timer_set(&data->timer, 0.01, interval, on_timer, data);
6     while(data->run)
7         sleep(1);
```

Listing 4.10: Simple Real-time scheduler

The `on_timer` function executes actors and deletes the timer after the dataflow graph is fully processed. Using this scheduler processing of an RTP stream takes exactly the time of the length of the stream. Using an additional actor for balancing jitter from the network, an IP stream could be directly processed after unwrapping the RTP packet.

4.4 PLC Subsystem

The following sections will describe functions of actors in the PLC subsystem. This model is designed to match description of G.711 recommendation appendix I [23]. Only important parts of the source code are referred to directly, the rest can be found in appendix B. Waveforms of the dataflow output are shown in appendix C.

4.4.1 Store and Forward Actor

A main functionality is to store a history of the signal (referred as `saf` actor). This actor is placed in the beginning of the subsystem, storing the incoming signal. But the actor also stores concealed packets to have history available for more packet loss which may happen. A reason for this is pictured in figure 4.8 where one packet is lost in the first place. After this first loss is concealed by parts of the previous frame one new frame is arriving. The next frame also filled with contents of the preceding. Another missing packet is generated differently than the first one: a longer section is factored

in, which can exceed the length of one frame. Therefore, concealed frames have to be available in the history buffer of a signal.

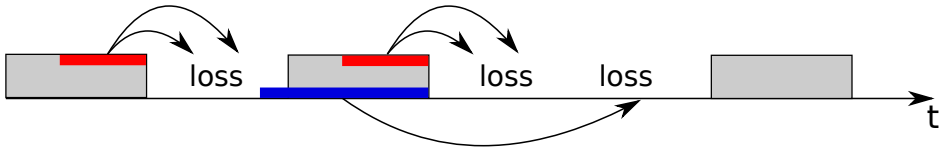


Figure 4.8: Concealment of multiple packet loss

This is done by a loop from the last actor (`overlap_add`) back to the `saf` actor (see figure 4.3). A token is send only if a packet was concealed and has not been copied to the history buffer. Therefore, a input is called `in_gen`. Another input is also the input of the whole subsystem called `in_good`. All possible modes are shown in table 4.4.

	Input		Output		
Mode	in_good	in_gen	out_good	out_store	Comment
good	1	0	1	0	normal operation
loss	0	0	0	1	first loss
loss_store	0	1	0	0	second and more loss
good_loss_store	1	1	1	1	after last loss
good_store	1	1	1	0	two after last loss

Table 4.4: Available modes in the `saf` actor

This actor guarantees that a defined length of the signal is available as history (total length: `store_size`). Definition is done at the initiation of the actor together with the determiniation of the values `samples_per_frame` and `bytes_per_sample`. In the following description of operation an assumption is made that $samples\_per\_frame = 80$ and $store\_size = 320$. In the normal mode ("good"), history is copied from position 80 to 0 with a length of 160. The new payload is copied to position 160 with the length of 80. Available "next modes" are only "good" or "loss" and no history is updated. As soon as a token is missing at the time of execution (mode "loss"), the whole buffer is passed on at output `out_store`. The setting is now $next\_mode = loss\_store|good\_loss\_store$. In the case of operating in "loss store" a token is copied in the same way as in mode "good" into the history buffer. If a packet loss is over (mode "good_loss_store"), the actor receives two tokens with payload data, one at each input. Therefore, the old

history is copied from position 180 to 0 with a length of 80. The payload of a token from `in_gen` is copied to position 80 and samples from `in_good` to 160, both with a length of 80. The "next mode" can be either "loss" or "good_store" before returning to normal operation in the mode "good". Figure 4.9 shows possible "next modes" in a flow diagram.

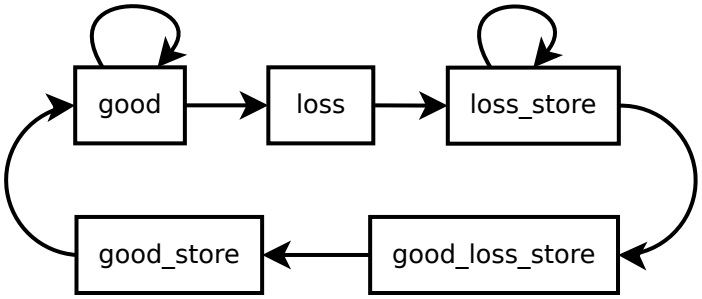


Figure 4.9: Next mode specification of `saf` actor

4.4.2 Generate Synthetic G.711 Actor

This actor implements the core functionality of the PLC algorithm based on the G.711 recommendation of appendix I (see [23]). It is described in section 2.3.2 and only actor specific behaviour and mode properties will be further explained in the following paragraphs.

As shown in figure 4.8 actor `gen_syn_g711` will be only active if tokens of a `saf` actor are put into input FIFO `in`. If no RTP packages are lost the actor will never be executed. As soon as a token arrives the payload is copied into the actors context. Pitch calculations are made and relevant samples are copied into a new packet. No overlap is done in this actor. The "next_mode" is either "create_follow" or "create_last". In the case of "create_follow" the same buffer is used and no input tokens are processed. The pitch will be different on multiple executions and volume of samples will get lowered. After five concealed samples only silence is created (mode "create_zeros"). Another token at the input FIFO marks the end of activity and one last concealed packet is created. Afterwards the actor returns to an inactive mode until a next activity starts with another input token. Table 4.5 gives an overview over production and consumption

rates in different modes.

Mode	in	out	Comment
inactive	0	0	default
create_first	1	1	after one token
create_follow	0	1	after two to five tokens
create_zeros	0	1	after more than five tokens
create_last	1	1	after the last lost packet

Table 4.5: Available modes in the `gen_syn_g711` actor

The actor contains only source code to find a pitch and create a new packet with multiple pitch periods. This makes it very simple to change the logic of this actor and evaluate slightly different implementation.

4.4.3 Overlap Add Actor

The PLC algorithm delays the signal by 3,75 ms (equals 30 samples) to allow a smooth transition between original and concealed packages (see 2.3.2). But LWEIDF as a programming paradigm does not support the dimension of time. Therefore, the actor implements a small buffer which has the $length = 80+30 = 110$. The incoming payload is copied to position 30, before samples from position 80 are copied to 0 with the length of 30. The outgoing payload is taken at position 0. Placing this actor between a read and a write actor it results in 30 samples silence in the beginning of the stream and a length extension of 30 samples. The behaviour is pictured in figure 4.10 a.

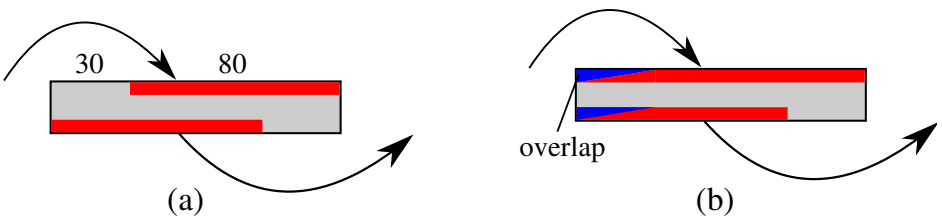


Figure 4.10: Functionality of the `overlap_add` actor

In case of a packet loss the actor changes the mode to “write_gen” and the token from “in_gen” is consumed which can be longer than 80 samples (see 4.10 b). Only the last

80 samples are copied to position 30, the other samples are overlaped with existing samples in the buffer (blue/red). The exact length of the token from `in_gen` depends on the pitch position which is calculated by the `syn_gen_g711` actor. In the "write_gen" mode the actor can rather stay in this mode or switch to "mix" if an input token is in the FIFO `in_good`. It consumes token from both inputs and overlaps it up to the length of a whole packet. In both modes "write_gen" and "mix" a token is created and send to the edge `out_store` to maintain a valid history in the saf actor. After this the actor returns to the "write_good" mode. All available modes can be seen in table 4.6.

	Input		Output		
Mode	in_good	in_gen	out	out_store	Comment
init	0	0	0	0	initialization
write_good	1	0	1	0	normal operation
write_gen	0	1	1	1	during packet loss
mix	1	1	1	1	after packet loss

Table 4.6: Available modes in the `overlap_add` actor

In the "init" mode the entire buffer is initiated with silence. The following diagram shows the possible "next modes" (figure 4.11). It can be seen that "init" mode is only executed once and the actor continues to process tokens as long as tokens are in the input FIFOs.

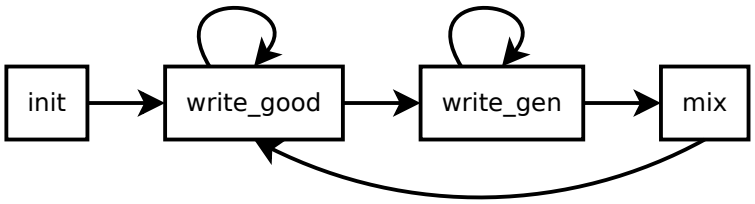


Figure 4.11: Next mode specification of `overlap_add` actor

4.4.4 Effective scheduling for the subsystem

A dataflow can be scheduled with canonical scheduling which comes with the LWDF API. Using real time scheduling is also possible. The model in figure 4.3 consists

of three actors, two of them have to be invoked for processing of data. The third actor, `gen_syn_g711`, is rarely invoked, but the enabling function is called on every single execution of the dataflow. Customized scheduling can lower the needed system resources. This is done by eliminating the for-loop which executes all actors in the scheduler and inserting this code instead (listing 4.11 - simplified code):

```
1 do {
2     progress = 0;
3     progress |= lide_c_util_guarded_execution(actors[READ]);
4     progress |= lide_c_util_guarded_execution(actors[SAF]);
5     if (actors[SAF]->mode!=SAF_MODE_GOOD)
6         progress |= lide_c_util_guarded_execution(actors[SYN_GEN_G711]);
7     progress |= lide_c_util_guarded_execution(actors[OVERLAP_ADD]);
8     progress |= lide_c_util_guarded_execution(actors[WRITE]);
9 } while (progress);
```

Listing 4.11: Customized PLC scheduler

Function `lide_c_util_guarded_execution()` is only called if the `saf` actor is exectured in a mode where it has to conceal a packet loss. Otherwise the enable function is not called by an actor execution function.

4.4.5 Implementing other PLC algorithms

Since the development of the G.711 algorithm and its PLC implementation many improved versions have been implemented. Through the segmentation of the functionality in the separate actors, similar PLC algorithms can be implemented with simple changes of the settings or in single actors. An example would be the algorithm described in [30]. It uses a different history buffer size and only an overlap size of 10%.

4.5 VAD Subsystem

Many actors in the VAD subsystem implement a very basic and independent behaviour. They are not mainly dealing with RTP packets but with signal processing algorithms

like calculating a signal spectrum. Some of the actors are only implemented as prototypes and need to be completely implemented in a next development phase.

4.5.1 Preprocess Actor

This actor handles forwarding of packages and payloads. It distributes data to five edges. One of the FIFOs receives a whole RTP packet, all other FIFOs only get the payload of a RTP package. The actor operates only in one mode, the consumption rate is $X_1 = 1$ which results in an input vector $I_a = X_1$. The production rate is $O_a = (1 \times 1 \times 1 \times 1 \times 1)$. If no tokens are available at input edge `in`, the enable function returns "false" and this actor wont be invoked.

General signal processing on a payload such as filters should be done previous to this actor. An example could be a high pass filter. Specific operations on a payload which are only needed on a single output should be done after the actor. Only processing which would be done redundant could be placed in or before the preprocessing actor.

4.5.2 Signal Characteristics Actors

In order to decide if a payload of a packet should be considered voice or noise, different signal characteristics have to be calculated. The modelled VAD algorithm (see figure 4.4) uses the values of signal energy, low-band energy, number of zero crossings and spectral data. The last one will be transferred as an array which is filled with values of respective frequency bands. The size of the array is defined during initialization. Other values are put directly in the FIFO. All actors have only one mode and a constant consumption and production rate of "1".

Calculations on the characteristics of the audio signal have to be implemented depending on the used hardware. This can be a DFT to calculate the signal energy or a simple multiplication in combination with additions. For implementing a prototype the second method was used. Source code can be found in appendix B.

To create a payload for CNG the signal energy is needed. To create a noise on the

receiver's side which is similar to the removed noise at the sender's side spectral information can be transferred, too (see 2.1.2). Therefore, `energy` and `spectral` actor have two outputs, one fires tokens in an edge to a VAD algorithm and the other gives tokens in an edge to an CNG encoder.

4.5.3 VAD Algorithm Actor

Actor `vad-algorithm` is initialized with a numbers of spectral information, with an initial "last VAD decision" defined as "true" in mode "active". The four inputs have to have one token in their FIFO otherwise this actor cannot be enabled. Values are processed according to recommendations in G.729 annex II (see 2.4). If the result of the algorithm is a determination of voice, a token with a value "true" is created for all output edges which are connected to `vad-algorithm`. Otherwise the output token to `dtx-algorithm` contain "false", actor `vad_encoder` will not receive any token. Further processing of this information will be discussed in the following sections.

4.5.4 CNG Encoder Actor

This actor creates a packet with a CNG payload. All input edges have to be filled with one token in order for the actor to be enabled. The only mode is "active" and production rate is "1". The first byte of the payload is taken from the `in_energie` edge, following bytes come from the `in_spectral` FIFO. The exact number of bytes which is provided by the initialization routine of this actor. This newly generated payload is already wrapped in an RTP structure and forwarded to a DTX algorithm.

4.5.5 DTX Algorithm Actor

This actor decides when it is necessary to send an RTP packet with a CNG payload. Its behavior is based on the following two ideas which can also be mixed:

- CNG payload is sent cyclic. This could be for example every 200ms and would guarantee a regular update. If noise characteristics change faster, it would not

have any effect. In a case of no change the packet can be considered as useless overhead.

- CNG payload is send when the noise characteristics change. This is efficient in regard to the transmission volume but more costly in terms of calculations which have to be done.

Whatever implementation is done for a DTX actor, the structure will stay the same. Two modes have to be implemented, one in which a packet is send and another which only consumes tokens. Input `in_vad` is interpreted in order to send only CNG payload packages if needed. It also guarantees that a CNG payload packet is send in the beginning of a noise-period. Table 4.7 gives an overview over the two modes of this actor.

	Input		Output	
Mode	in_vad	in_dtx	out	Comment
active	1	1	0	unchanged CNG values
active_out	1	1	1	send CNG packet

Table 4.7: Available modes in the `dtx_algorithm` actor

A prototype of this actor is implemented with a cyclic transmission of packets with CNG payload. The constructor function is called with a parameter for a rate of package transmission.

4.5.6 VAD Encoder Actor

This actor creates the final output of the whole subsystem. Three important tokens come together at this point:

- original RTP packet (edge: `in`)
- decision voice or noise (edge: `in_vad`)
- RTP packet with CNG payload (edge: `in_cng`)

In the initial mode "voice" incoming packets are forwarded and `in_vad` edge is emptied. Possible "next modes" are all three available ones because modes are only depending on

inputs and not on a sequence of actor executions. In case of a "noise" decision (no input token on `in_vad`) and an available `in_cng` token, a CNG payload is used to replace the original payload in the RTP packet coming from `in`. If a CNG payload token is absent, no RTP packages are issued. Table 4.8 shows possible modes with the rate of consumption and production tokens.

	Input			Output	
Mode	in	in_vad	in_cng	out	Comment
voice	1	1	0	1	forwarding voice packet
noise_info	1	0	1	1	forwarding CNG payload packet
noise	1	0	0	0	dumping voice packet

Table 4.8: Available modes in the `vad_encoder` actor

Only the original RTP packet from `in` contains a correct sequence number and timestamp. Therefore, this header information is always used and can be modified later. This is done when a CNG payload is taken from `in_cng` and the payload type has to be changed.

4.5.7 Effective scheduling for the subsystem

Scheduling can be enhanced similar to the approach which was used for the PLC dataflow (see section 4.4.4). In some cases results of enabling functions ($B = (true|false)$, see 3.2) of an actor can be derived from modes of previous actors. This can be done when it comes to optimization of scheduling in the VAD subsystem. Running the dataflow in a single core environment actor `vad-algorithm` would be executed before the `cng-encoder`. By this time the decision is known and actor `cng-encoder` could be invoked in a mode in which it just consumes tokens and drops them. This minimizes the usage of the calculations. Actor `dtx-algorithm` could even be not invoked at all. Using this scheduling, VAD algorithms performs all calculations which are necessary for a VAD decision but in case of no CNG encoding corresponding parts of the dataflow are not executed.

4.5.8 Implementing other VAD algorithms

There are multiple signal characteristics which can be used to determine if an audio stream is voice or noise. Newer algorithms use different features of a signal to make this decision. A proposed real time algorithm in [36] uses energy (E), Spectral Flatness Measure (SFM) and the most dominant frequency component of a speech frame spectrum (referred to as F). To implement this algorithm in a dataflow, new actors would have to be designed which calculate values on which the decision algorithm is based. The actor which makes the VAD decision would have to be duplicated and modified but the rest of the subsystem could be adopted.

Another algorithm [21] uses divergence between the long term spectral envelope (LTSE) in combination with a signal to noise ratio (SNR) to determine if a packet is noise (resp. silence) or voice. Through implementing actors for this calculations and changes of the decision actor this algorithms behaviour could be implemented.

Even other technics are used to determine the signals importance: neuronal networks ([1]) and support vector machines ([25]). This functionality could be implemented in new sub-subsystems and replace the actor `vad-algorithm`. Actors which calculate signal characteristics for a CNG payload have to be maintained but others could be eliminated.

4.6 CNG Subsystem

Following sections are giving a detailed view of the CNG subsystem actors. As mentioned in the modelling section, conversion actors can be inserted which also process RTP packets. Results of a dataflow execution can be looked at in appendix C.

4.6.1 VAD Decoder Actor

This actor is a counterpart to the VAD encoder: it sorts packets by payload type (PT). It has one input and three outputs. The first output fires tokens containing voice information and the second one forward packets with PT 13. When this actor receives

the first voice packet it is equipped with a special flag (see section 2.4). The actor needs only one mode to operate in because it is not always firing. If it receives no token, it will not be invoked.

4.6.2 CNG Decoder Actor

Actor `cng-encoder` is the core of the subsystem. It generates CNG packets based on the given differences of signal energy and spectral information. After initiating the actor mode is set to "generate_set". As long as no input token is provided the actor will not be enabled. With the first token the actor reads all information about current signal characteristics and stores it in an internal buffer. Valid "next modes" are now (equation 4.4):

$$next\_mode = generate\_set|generate|inactive \tag{4.4}$$

Mode "generate_set" is valid because signal information can be updated any time. Tokens can be generated with the same characteristics, therefore also mode "generate" can be invoked. In case of a voice packet which is received by `vad-decoder` and a token in FIFO `in_terminate`, the CNG decoder will switch in an inactive mode.

This mode will set "generate_set" as a "next mode" and with another token containing signal information the actor will be enabled again and produce CNG packets. Table 4.9 shows a complete overview of modes and consumption resp. production of tokens.

	Input		Output	
Mode	in	in_terminate	out	Comment
generate_set	1	0	1	generate packet based on new data
generate	0	0	1	generate packet based on stored data
inactive	0	1	0	disable generation of packets

Table 4.9: Available modes in the `cng_decoder` actor

4.6.3 MUX Actor

Actor `mux` allows a subsystem to create a single output. Depending on available input tokens it switches to the proper mode to consume tokens, therefore all three modes are valid modes for a "next mode". In case of a voice and a noise input, the first input is preferred. If no tokens are available, the actor will not be executed.

	Input		Output	
Mode	in_voice	in_noise	out	Comment
voice	1	0	1	get and forward voice packet
noise	0	1	1	get and forward noise packet
both	1	1	1	receives both tokens and only forwards voice

Table 4.10: Available modes in the `mux` actor

4.6.4 Implementing other CNG algorithms

A `cng-decoder` actor can only process one specific format of signal characteristics. It is always bounded to the data which is created by a matching `cng-encoder` actor. If five spectral information are encapsulated in a CNG payload these five have to be handled. This results in a variety of operations: e. g. twelve spectral information and an output in G.711 or six spectral information and a linear output. Creating multiple actors for different usages is better fitting in this case than creating multiple modes. The main reason is that functionalities will not be switched during the operation of a dataflow. Another reason are different output codecs which could possibly change the modelling of the dataflow. Separation of voice and CNG packages has to be performed in the same way as multiplexing of tokens. Therefore, subsystems help both designers and actor programmers to implement new CNG algorithms with a modification of the core functionality only.

4.6.5 Effective scheduling for the subsystem

Due to processing every packet a CNG decoder will get invoked even if the system processes a voice packet. This it not necessary and will have a negative influence on

the performance of a subsystem. A possible solution would be to introduce two modes for the VAD decoder, both performing the same actions. Depending on the input token, "next mode" would be either voice or noise, corresponding with the actual type of packet. Before the next actor (`cng_decoder`) is enabled modes of `vad_decoder` are checked. Only if it is noise, the actors enable function would be called. A similar implementation for a scheduler has been shown for the PLC algorithm (see 4.4.5).

4.7 Applying non-deterministic behaviour to TDIF

Just as LWDF, the TDIF approach is based on CFDF which allows only deterministic behaviour. A major advantage of TDIF is a language which allows definition of actors and dataflows in the DIF format. A simple RTP application has been implemented with TDIF and an extension in the enable function is proposed to support non-deterministic behaviour.

4.7.1 Sample based dataflow

Read and write actors in LWEIDF 4.3.2 have been specially designed for RTP applications. They read a whole packet of samples from a file or an equivalent data stream. In TDIF implementation another approach has been made: single samples are read by a wave reader and put into a FIFO. As soon as enough samples are collected from a RTP packet the actor `rtp_packetizer` is consuming a defined number of tokens. In an example shown in figure 4.12 one packet is transferred to `rtp_depacketizer` which fires every single sample to the `wav_writer`.

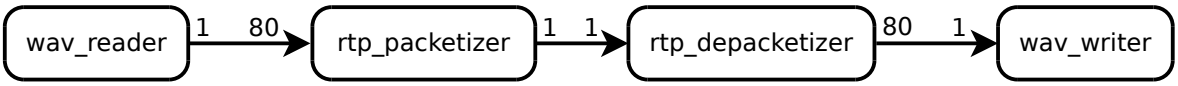


Figure 4.12: Simple TDIF dataflow for RTP packet transmission

A main advantage of this approach is that peripheral inputs of hardware (e.g. from a codec) can be used as a direct input to the dataflow without any preprocessing.

4.7.2 RTP actors

The first step in developing a TDIF actor is a definition of actors with keywords of the TDIF language. To define the `rtp_packetizer` actor matching notation can be found in listing 4.12.

```
1 input input1 short
2 output output1 void*
3 param size int
4 mode init
5 mode process
6 mode send
```

Listing 4.12: Definition of the `rtp_packetizer` actor in the TDIF language

This actor has an input for samples and an output for a reference to a packet. A param is the size of the RTP payload and the actor can operate three modes: “init”, “process” which reads the samples and “send” which sends a packet with a payload of buffered samples. Using a TDIF compiler header files are created and implementation can be done. It is defined that the “init” mode does not consume or produce any tokens. The “process” mode only consumes a token, “send” mode consumes and produces a token.

An implementation of the invoke function can be found in appendix B together with definitions and implementations for other actors. An execution of this dataflow results in the source file being similar to the file which is created as an output.

4.7.3 Proposal for a non-deterministic implementation

To allow a non-deterministic behaviour of TDIF the enable function has to be extended. It has to be able to select a mode based on available inputs of a set of modes. In contrast to LWEIDF, where the enable function is implemented manually, TDIF allows only a definition of consumption resp. production rate. The enable function itself is a part of the TDIF library. In order to change the behaviour of the general enable function it has to be adapted. Macros which were developed for LWEIDF are also used in the non-deterministic TIDF. Listing 4.13 shows a modified function. A detailed description can be found below.

```
1  int tdifc_ec_enable_check(tdifc_ec_pointer ec, tdifc_tc_pointer tc) {
2    [...]
3    input_count = tdifc_tc_input_count(tc);
4    mode = tdifc_ec_get_mode(ec);
5    for (i = 0; i < tdifc_ec_get_mode_count(ec); i++) {
6      if (CHECK_MODE(mode, i)) {
7        ret = 1;
8        for (j = 0; j < input_count; j++) {
9          population = tdifc_tc_input_populations(tc, i, j);
10         consumption_rate = ec->f_consumption_rates(j, ec, i);
11         if (population < consumption_rate)
12           ret = 0;
13       }
14       if (ret==1) {
15         tdifc_ec_set_mode(ec, i);
16         return 1;
17       }
18     }
19   }
20   return 0;
21 }
```

Listing 4.13: Proposed enable function for TDIF

The call of the original function (line 1) has not been modified. Definitions of variables have been skipped (line 2), a full function can be found in appendix B. Input count and a current set of modes are set using the get-functions from the TC and EC interface (line 3/4). A new for-loop is introduced which goes through all possible modes and checks if it applies to a set of given modes (line 5/6). A new function *tdifc_ec_get_mode_count* has to be implemented that returns a numbers of modes. Variable *ret* is set to "1" as a start value (line 7) and the for-loop of the original function checks all inputs if they match consumption resp. production rate (line 8-13). In case of no matches the *ret* variable is set to "0". Otherwise, the variable is still at its start value $ret = 1$. Then the current state of the actor matches the requirements of at least one given mode and the first one is set (line 14-17). Therefore, the first mode which should be tested for

matching should be defined with the lowest number. If it is not possible to enable any mode the actor is not enabled.

5

Conclusion

After the first experiments and developments it became clear that the algorithms (PLC, VAD and CNG) could not be implemented in any of the currently used dataflow approaches with the proposed rules of the APIs. The problem is that, even with the dynamic LWDF, the dataflow model is deterministic. This means that the mode in which the actors execute is defined before it is enabled. This is very useful, especially for optimized scheduling. But it creates a type of predefinition which makes it impossible to model and implement RTP algorithms.

5.1 Achievements

The EIDF model fits very well for the implementation of non-deterministic RTP algorithms. An API was only available for the deterministic CFDF and could not be applied to dataflow models with non-deterministic problems. Therefore, the definitions of EIDF have been used to propose a derived API called LWEIDF. It is able to handle a set of modes in the enable function and return a set of modes after the invoke function is called. In this way it supports the modelling of non-deterministic dataflows like the subsystems for PLC, VAD and CNG. As a prove of concept three algorithms have been modelled and implemented prototypically. Dataflow models implemented with LWEIDF now have an even broader range of usage.

Also TDIF can be based on the concept of EIDF and operate non-deterministic. A general enable function has been implemented to show the opportunities which arise

from this kind of implementation.

5.2 Advantages of dataflow implementations

Using dataflows for developing image- and signal processing applications is a great innovation for both hardware- and software developers. The implementation process is application based and complexity hidden in actors. Therefore, dataflow designers have a great overview and actors can be tested individually.

After the first steps of development testing is already possible and highly recommended. During the whole process of development actors and simple subsystems can be tested for side effects and handling errors.

An application design can be used for every platform and type of hardware. For a customized implementation only actors and FIFO's have to be developed. This makes it possible to use rapid prototyping and significantly shorten the time to market period [40].

5.3 Further Research and Developments

Current implementations of algorithms in EIDF with the LWEIDF-API are only done at the level of a prototype. First results can be viewed in appendix C. Therefore, performance tests could not be realized at this early stage of development. Further implementation and optimization will give a deeper insight into the opportunities of dataflow solutions for dynamic and non-deterministic problems. Another option would be the implementation of other IP and RTP specific routines to use only a DSP or a graphic engine to perform all necessary tasks. Modelling RTP application for video streaming applications is another challenging task. This is not meant to be an alternative to the object-oriented approach. In fact, both can be used together. One practical application could be the construction of libraries with classes and applications through connecting actor instances using dataflow techniques.

Bibliography

[1] A.Akbari, M.F..M..B..: *A Model-based Voice Activity Detection Algorithm using probabilistic neural networks.* In *Proceedings of APCC2008*, pp. 83 – 86, Tokyo, Japan, 2008.

[2] Ahmed, A., Madani, H., and Siddiqui, T.: *VoIP Performance Management and Optimization.* Networking Technology: IP Communications. Pearson Education, 2010 - ISBN 9780132583060.

[3] Bharath, S. and Arkansas, U. of: *Message Reliability Over UDP.* University of Arkansas, 2008 - ISBN 9780549666301.

[4] Bhattacharya, B. and Bhattacharyya, S.S.: *Parameterized dataflow modeling for DSP systems.* IEEE Transactions on Signal Processing, 49(10):2408–2421, October 2001.

[5] Bhattacharyya, S.S.: *Compiling Dataflow Programs for Digital Signal Processing.* PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, July 1994.

[6] Bhattacharyya, S.S., Murthy, P.K., and Lee, E.A.: *Synthesis of embedded software from synchronous dataflow specifications.* Journal of VLSI Signal Processing Systems for Signal, Image, and Video Technology, 21(2):151–166, June 1999.

[7] Bhattacharyya, S.S. et al.: *The DSPCAD integrative command line environment: Introduction to DICE version 1.1.* Technical Report UMIACS-TR-2011-10,

Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

[8] Bhattacharyya, S.S. et al.: *Using the DSPCAD integrative command-line environment: User's guide for DICE version 1.1.* Technical Report UMIACS-TR-2011-13, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

[9] Bhattacharyya, S.S.: *Lecture notes: Design and implementation of signal processing software.* Technical report, Institute for Advanced Computer Studies, University of Maryland at College Park, 2007.

[10] Chen, Y. et al.: *Signal processing on platforms with multiple cores: Part 1 — overview and methodologies.* IEEE Signal Processing Magazine, 26(6):24–25, November 2009. Guest Editors' Introduction.

[11] Chen, Y. et al.: *Signal processing on platforms with multiple cores: Part 2 — design and applications.* IEEE Signal Processing Magazine, 27(2):20–21, March 2010. Guest Editors' Introduction.

[12] Chen, Y., Liu, L., and Bhattacharyya, S.S.: *Guest editorial: Special issue on multi-core enabled multimedia applications & architectures.* Journal of Signal Processing Systems, 57(2):121–122, November 2009.

[13] Dennis, J., Fosseen, J., and Linderman, J.: *Data flow schemas.* In Ershov, A. and Nepomniaschy, V.A. (editors): *International Symposium on Theoretical Programming*, volume 5 of *Lecture Notes in Computer Science*, pp. 187–216. Springer Berlin / Heidelberg, 1974 - ISBN 978-3-540-06720-7. 10.1007/3-540-06720-5_15.

[14] Durkin, J.: *Voice Enabling the Data Network: H.323, MGCP, SIP, QoS, SLAs, and Security.* Networking Technology Series. Cisco Press, 2003 - ISBN 9781587050145.

[15] Fernandez, M.: *Models of Computation: An Introduction to Computability Theory.* Undergraduate Topics in Computer Science. Springer, 2009 - ISBN 9781848824331.

[16] Gu, R., Bhattacharyya, S.S., and Levine, W.S.: *Methods for efficient implementation of model predictive control on multiprocessor systems.* In *Proceedings of the IEEE International Conference on Control Applications*, pp. 1357–1362, Yokohama, Japan, September 2010.

[17] Hersent, O., Petit, J., and Gurle, D.: *IP telephony: deploying voice-over IP protocols.* John Wiley, 2005 - ISBN 9780470023594.

[18] Hiremath, M.: *Multimedia over ip - 03.* http://mrutyunjayahiremath.blogspot.co.at/2010/10/multimedia-over-ip-rtp.html (08/30/2012), oct 2010.

[19] Hsu, C.: *Dataflow Integration and Simulation Techniques for DSP System Design tools.* PhD thesis, Department of Electrical and Computer Engineering, University of Maryland, College Park, April 2007.

[20] Hsu, C. et al.: *Dataflow interchange format: Language reference for DIF language version 1.0, user's guide for DIF package version 1.0.* Technical Report UMIACS-TR-2007-32, Institute for Advanced Computer Studies, University of Maryland at College Park, June 2007. Also Computer Science Technical Report CS-TR-4871.

[21] Iker Luengo, Eva Navas, I.O.I.S.I.H.I.S.D.E.: *Modified LTSE-VAD algorithm for applications requiring reduced silence frame misclassification.* In *LREC 2010 Proceedings*, pp. 1539 – 1544, Sliema, Malta, 2010.

[22] ITU-T: *Annex B: A silence compression scheme for G.729 optimized for terminals conforming to Recommendation V.70*, 1996.

[23] ITU-T: *Appendix I: A high quality low-complexity algorithm for packet loss concealment with G.711*, 1999.

[24] Jackson, M.: *Cisco predicts massive quadruple jump in global internet traffic by 2015.* http://www.ispreview.co.uk/story/2011/06/01/cisco-predicts-massive-quadruple-jump-in-global-internet-traffic-by-2015.html (08/30/2012).

[25] Jo, Q.H. et al.: *Statistical model-based voice activity detection using support vector machine*. Signal Processing, IET, 3(3):205 –210, may 2009 - ISSN 1751-9675.

[26] JungWon, J. et al.: *Streamlined embedded technologies*. Technical report, Institute for MBA Studies, University of Maryland at College Park, 2012.

[27] Kapilow, D.A.: *Method and apparatus for performing packet loss or frame erasure concealment*, Patent, 05 2006. US 7047190.

[28] Kausar, N. and Crowcroft, J.: *General conference control protocol*. In *Telecommunications, 1998. 6th IEE Conference on (Conf. Publ. No. 451)*, pp. 143 – 150, Edinburgh, UK, GB, Mar-Apr 1998.

[29] Khanna, G. et al.: *Voice Activity Detection for VoIP-An Information Theoretic Approach* . In *Global Telecommunications Conference, 2006. GLOBECOM '06. IEEE*, pp. 1–6, San Francisco, CA, USA, Nov-Dec 2006.

[30] Kovesi, B. and Ragot, S.: *A low complexity packet loss concealment algorithm for itu-t g.722*. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pp. 4769 –4772, 31 2008-april 4 2008.

[31] Kozierok, C.: *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*. No Starch Press Series. No Starch Press, 2005 - ISBN 9781593270476.

[32] Kuo, S., Lee, B., and Tian, W.: *Real-time digital signal processing: implementations and applications*. John Wiley, 2006 - ISBN 9780470014950.

[33] Lazzaro, J.: *Framing Real-time Transport Protocol (RTP) and RTP Control Protocol (RTCP) Packets over Connection-Oriented Transport*. RFC 4571 (Standard), July 2006.

[34] Li, A.: *RTP Payload Format for Generic Forward Error Correction*. RFC 5109 (Proposed Standard), Dec. 2007.

[35] Lublinerman, R. and Tripakis, S.: *Translating data flow to synchronous block diagrams*. In *In Proceedings of the IEEE workshop on*, 2008.

[36] Moattar, M.H. and Homayounpour, M.M.: *A simple but efficient real-time voice activity detection algorithm.* In *17th European Signal Processing Conference (EUSIPCO 2009)*, pp. 2549 – 2553, Glasgow, Scotland, 2009.

[37] Morlat, S.: *ortp, a real-time transport protocol (rtp,rfc3550) library.* http://www.linphone.org/eng/documentation/dev/ortp.html (08/20/2012).

[38] Perkins, C.: *RTP: audio and video for the internet.* Kaleidoscope series. Addison-Wesley, 2003 - ISBN 9780672322495.

[39] Perkins, C. et al.: *RTP Payload for Redundant Audio Data.* RFC 2198 (Proposed Standard), September 1997.

[40] Plishker, W. et al.: *Functional DIF for rapid prototyping.* In *Proceedings of the International Symposium on Rapid System Prototyping*, pp. 17–23, Monterey, California, June 2008.

[41] Plishker, W. et al.: *Heterogeneous design in functional DIF.* In Stenström, P. (editor): *Transactions on High-Performance Embedded Architectures and Compilers IV*, volume 6760 of *Lecture Notes in Computer Science*, pp. 391–408. Springer Berlin / Heidelberg, 2011.

[42] Raake, A.: *Speech quality of VoIP: assessment and prediction.* Wiley, 2006 - ISBN 9780470030608.

[43] Radojevic, I. and Salcic, Z.: *Embedded Systems Design Based on Formal Models of Computation.* Embedded Systems Series. Springer, 2011 - ISBN 9789400715936.

[44] Rosenberg, J. and Schulzrinne, H.: *An RTP Payload Format for Generic Forward Error Correction.* RFC 2733 (Proposed Standard), December 1999. Obsoleted by RFC 5109.

[45] Saha, S., Puthenpurayil, S., and Bhattacharyya, S.S.: *Dataflow transformations in high-level DSP system design.* In *Proceedings of the International Symposium on System-on-Chip*, pp. 131–136, Tampere, Finland, November 2006. Invited paper.

[46] Schulzrinne, H. and Casner, S.: *RTP Profile for Audio and Video Conferences with Minimal Control*. RFC 3551 (Standard), July 2003.

[47] Schulzrinne, H. et al.: *RTP: A transport protocol for real-time applications*. IETF Request for Comments: RFC 3550, July 2003.

[48] Sen, M. et al.: *Reconfigurable image registration on FPGA platforms*. In *Proceedings of the IEEE Biomedical Circuits and Systems Conference*, pp. 154–157, London, UK, November 2006.

[49] Shen, C., Plishker, W., and Bhattacharyya, S.S.: *Dataflow-based design and implementation of image processing applications*. Technical Report UMIACS-TR-2011-11, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

[50] Shen, C. et al.: *A lightweight dataflow approach for design and implementation of SDR systems*. In *Proceedings of the Wireless Innovation Conference and Product Exposition*, pp. 640–645, Washington DC, USA, November 2010.

[51] Shen, C. et al.: *The DSPCAD lightweight dataflow environment: Introduction to LIDE version 0.1*. Technical Report UMIACS-TR-2011-17, Institute for Advanced Computer Studies, University of Maryland at College Park, 2011.

[52] Shen, C. et al.: *A design tool for efficient mapping of multimedia applications onto heterogeneous platforms*. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, Barcelona, Spain, July 2011.

[53] Spurgeon, C.: *Ethernet: The Definitive Guide*. Definitive Guide Series. O'Reilly, 2000 - ISBN 9781565926608.

[54] The Maryland DSPCAD Research Group: *DSPCAD laboratory website*. http://www.ece.umd.edu/DSPCAD/home/dspcad.htm (08/30/2012).

[55] Zopf, R.: *Real-time Transport Protocol (RTP) Payload for Comfort Noise (CN)*. RFC 3389 (Proposed Standard), Sept. 2002.

Abbreviations

DSP Digital Signal Processor

CAD Computer-aided Design

API Application Programming Interface

DICE The DSPCAD Integrative Command Line Environment

LIDE Lightweight Dataflow Environment

SDF Synchronous Dataflow

CSDF Cyclo-Static Dataflow

PSDF Parameterized Synchronous Dataflow

PDF Parameterized Dataflow

EIDF Enable Invoke Dataflow

CFDF Core Functional Dataflow

LWDF Lightweight Dataflow

LWEIDF Lightweight Enable Invoke Dataflow

DIF Dataflow Interchange Format

TDIF Targeted Dataflow Interchange Format

EC Execution Context

TC Topological Context

FPGA Field-Programmable Gate Arrays

FIFO First In First Out

ITS Individual Test Subdirectory

ADT Abstract Data Type

IP Internet Protocol

WWW World Wide Web

RTP Real-time Transport Protocol

PT Payload Type

RTCP RTP Control Protocol

VOIP Voice over IP

PSTN Public Switched Telephone Network

SIP Session Initiation Protocol

PCMU Pulse Code Modulation μ-law

PCMA Pulse Code Modulation a-law

GSM Global System for Mobile Communications

ITU International Telecommunication Union

MSB Most significant bit

WAN Wide Area Network

LAN Local Area Network

TCP Transmission Control Protocol

UDP User Datagram Protocol

PLC Packet Loss Concealment

FEC Forward Error Correction

LBR Low-Bitrate Redundany

VAD Voice Activity Detection

CNG Comfort Noise Generation

DTX Discontinuous Transmission

LFS Line Spectral Frequencies

SID Silence Insertion Descriptor

SFM Spectral Flatness Measure

LTSE Long Term Spectral Envelope

SNR Signal-to-noise Ratio

Appendix

A

Dataflow Models

The following figures show the modelled RTP subsystems together with the test applications.

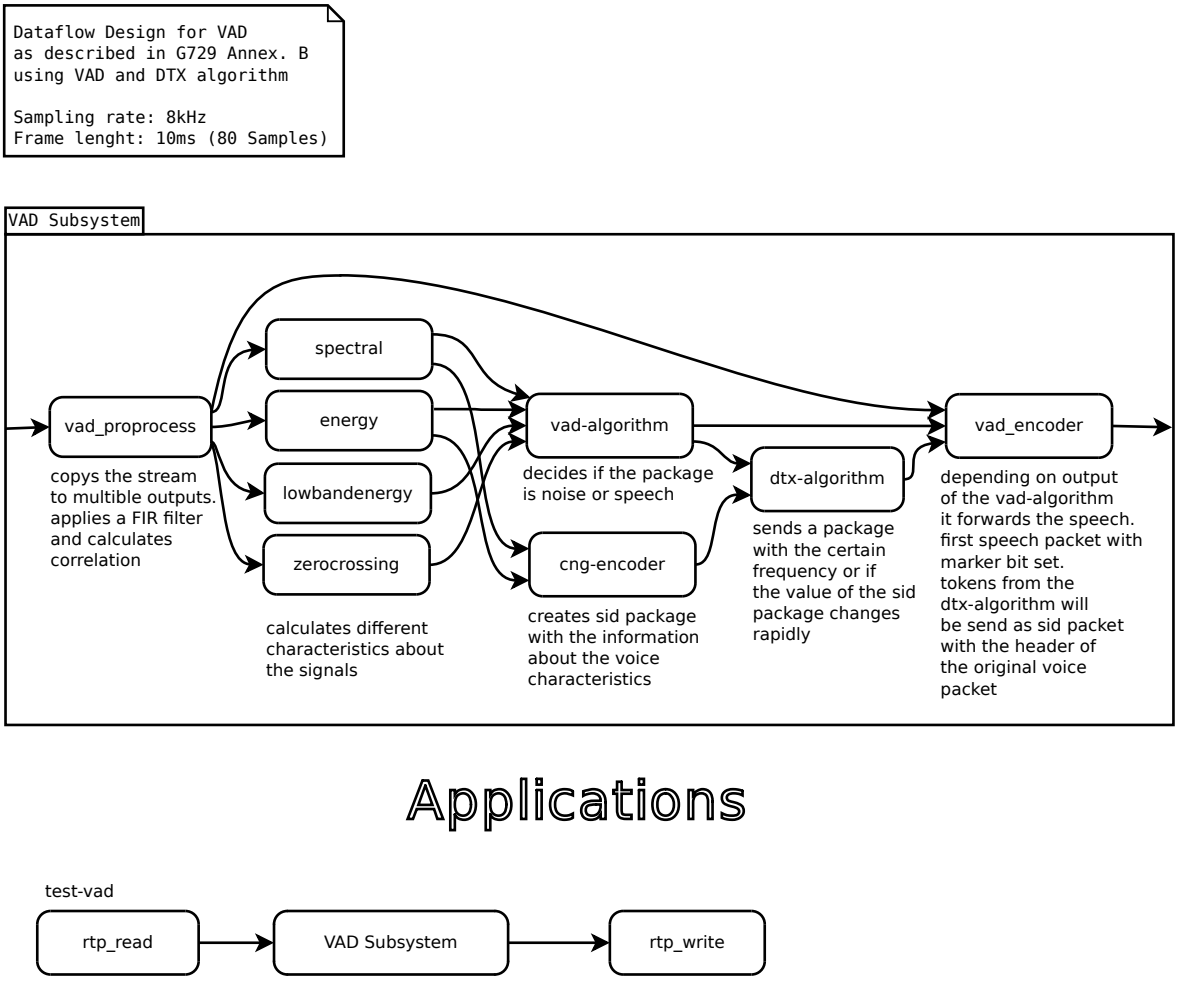
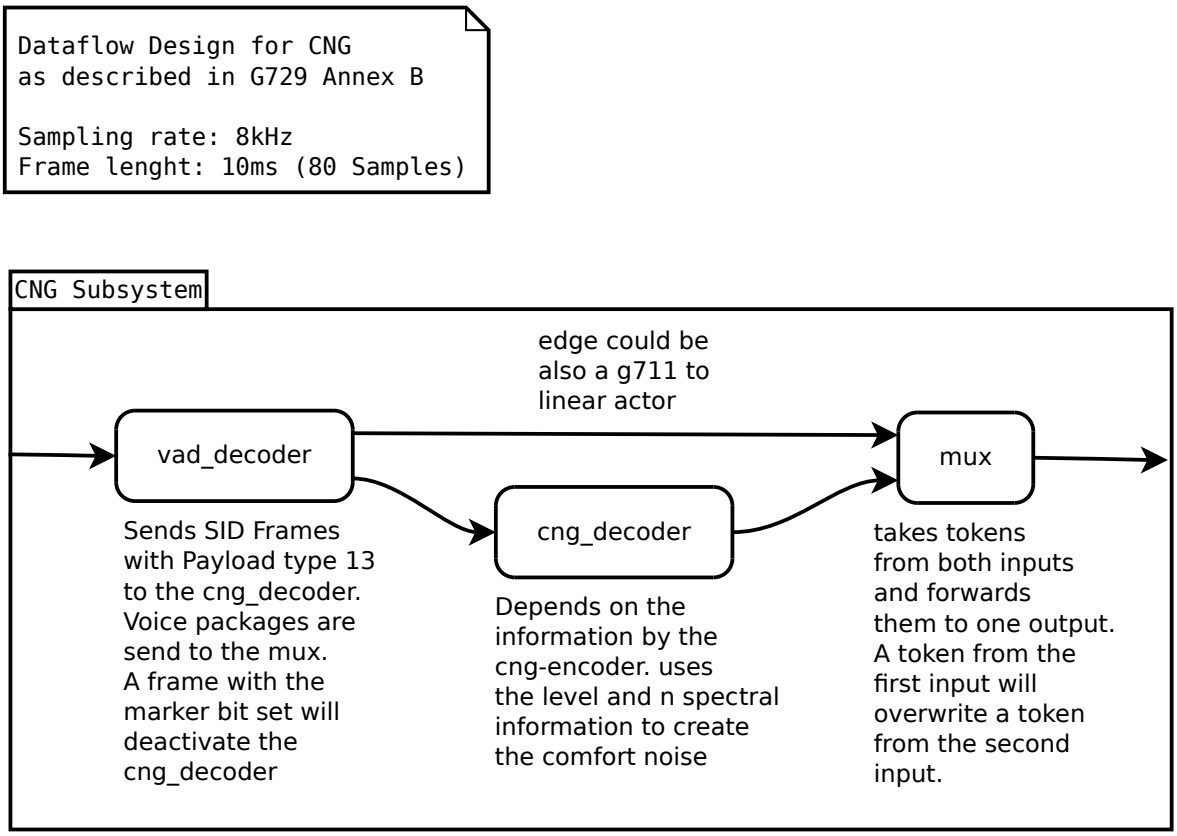


Figure A.1: Modelling VAD G.729 with applications



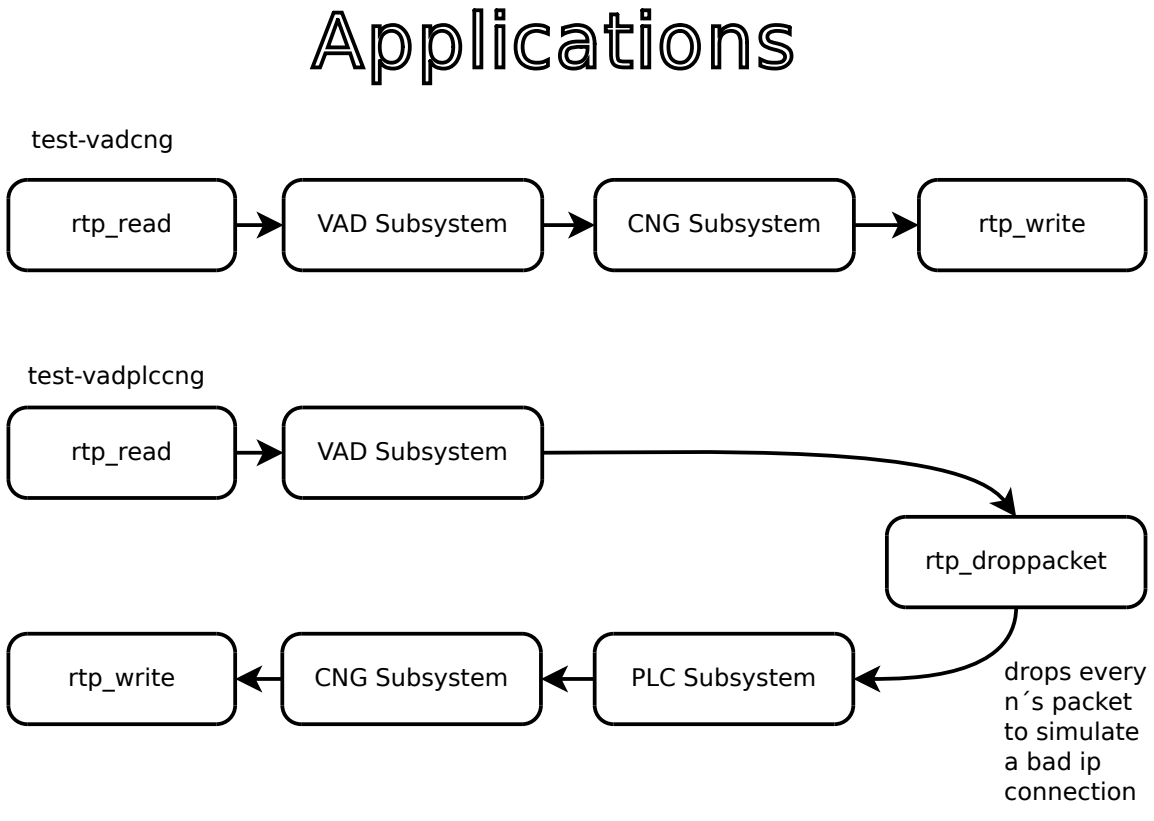
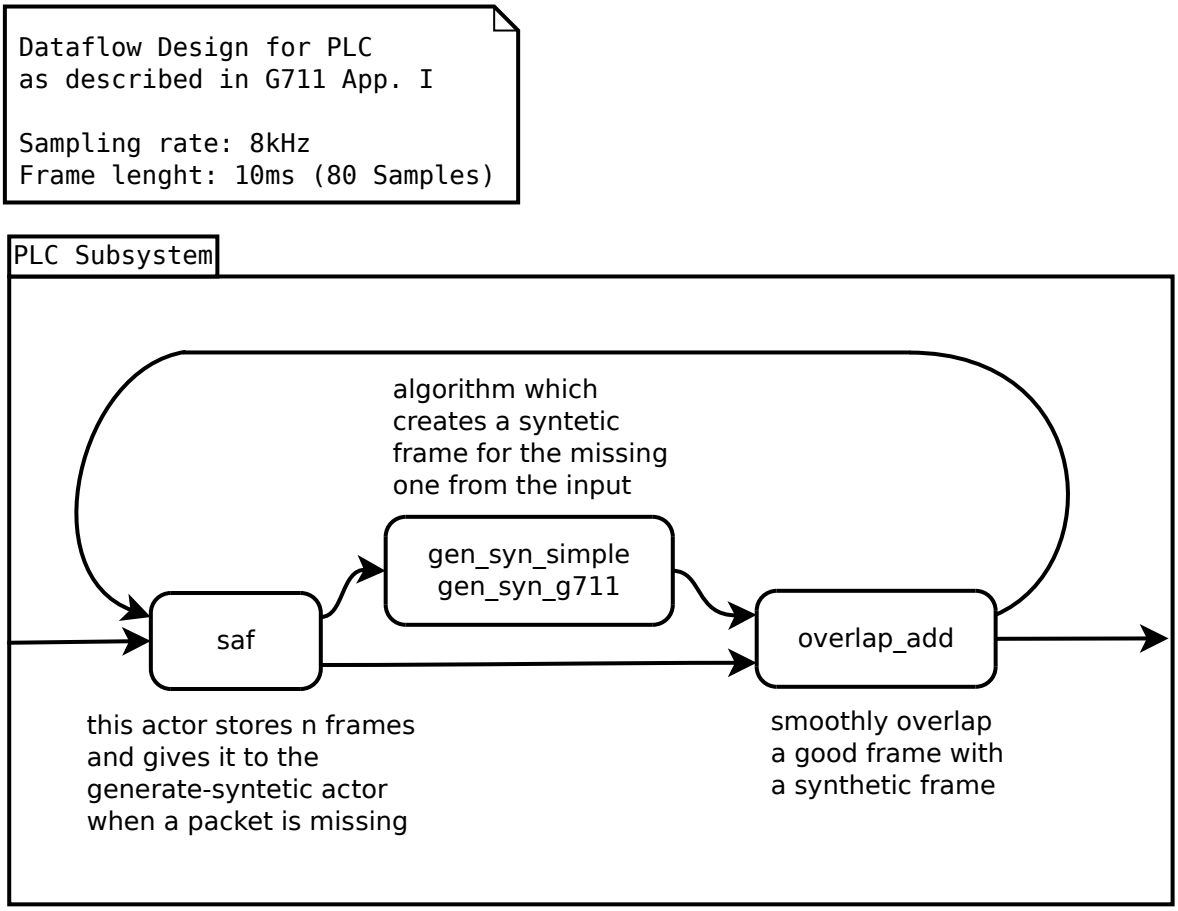


Figure A.2: Modelling CNG G.729 with applications



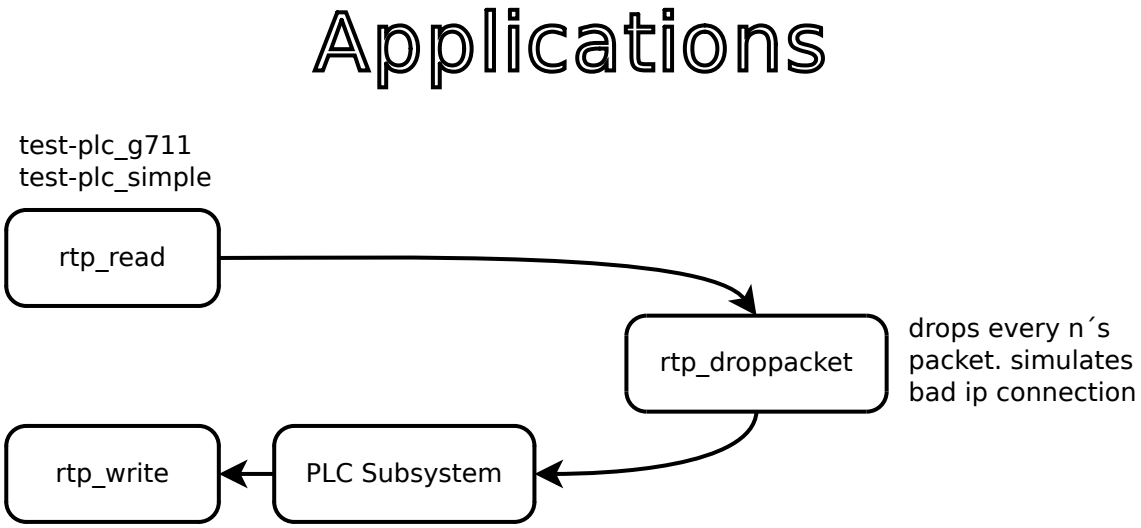


Figure A.3: Modelling PLC G.711 with applications

B

Sourcecode

The project sourcecode can be found on the enclosed CD, requested from the author or downloaded from the website of the DSPCAD Research group, referenced in [54].

```
1  int tdifc_ec_enable_check(tdifc_ec_pointer ec, tdifc_tc_pointer tc) {
2    int i = 0;
3    int consumption_rate = 0;
4    int input_count = 0;
5    int population = 0;
6    int mode = 0;
7    input_count = tdifc_tc_input_count(tc);
8    mode = tdifc_ec_get_mode(ec);
9    for (i = 0; i < tdifc_ec_get_mode_count(ec); i++) {
10     if (CHECK_MODE(mode, i)) {
11       ret = 1;
12       for (j = 0; j < input_count; j++) {
13         population = tdifc_tc_input_populations(tc, i, j);
14         consumption_rate = ec->f_consumption_rates(j, ec, i);
15         if (population < consumption_rate)
16           ret = 0;
17       }
18       if (ret==1) {
19         tdifc_ec_set_mode(ec, i);
20         return 1;
21       }
22     }
23   }
24   return 0;
25 }
```

Listing B.1: Full proposed enable function for TDIF

C

Dataflow generated waveforms

Outputs of the subsystems presented in appendix A are pictured below. Actors are not fully developed and the result is meant to be a prove of concept rather than an improvement of the outcome.

Figure C.1 contains three waveforms: The top shows the original signal, the one in the middle displays the effects of packet loss and a concealed result is pictured at the bottom.

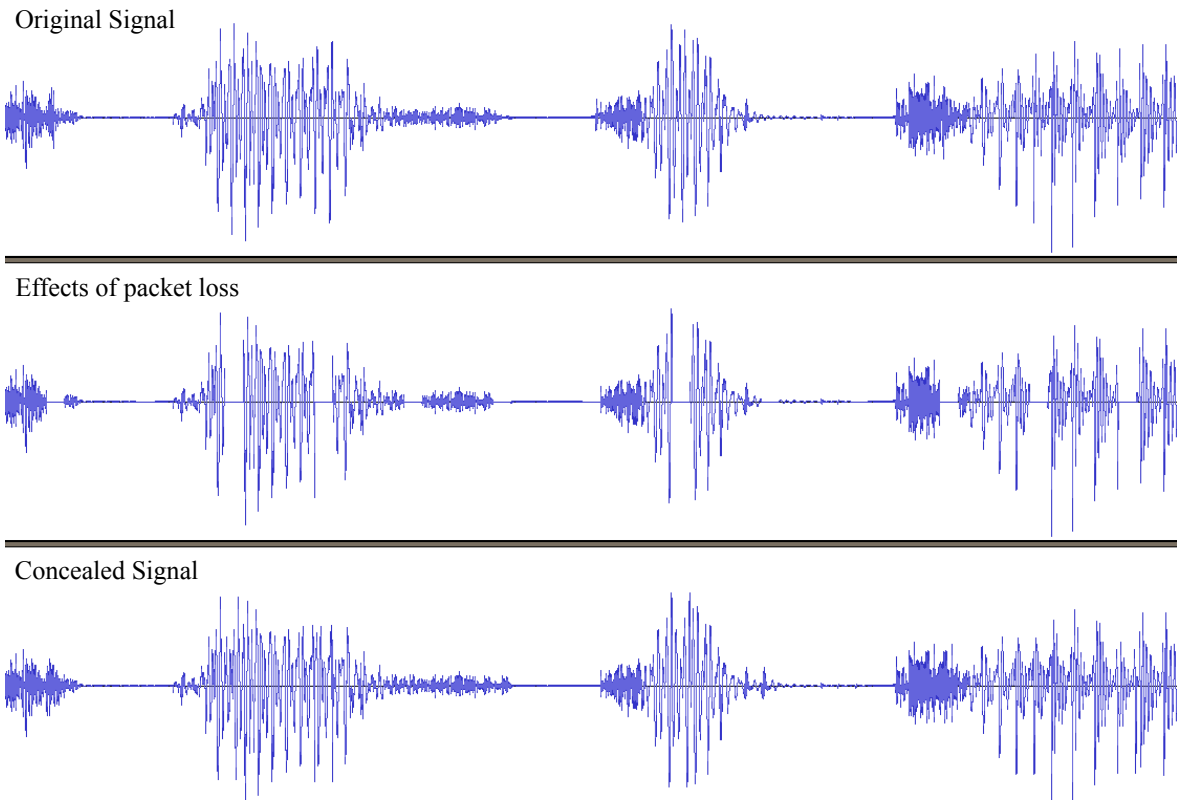


Figure C.1: Waveform effects of PLC

Effects of VAD and CNG are shown in figure C.2 : At first the orignal signal is presented, below noise passages are subpressed and the generated comfort noise is displayed at the bottom.

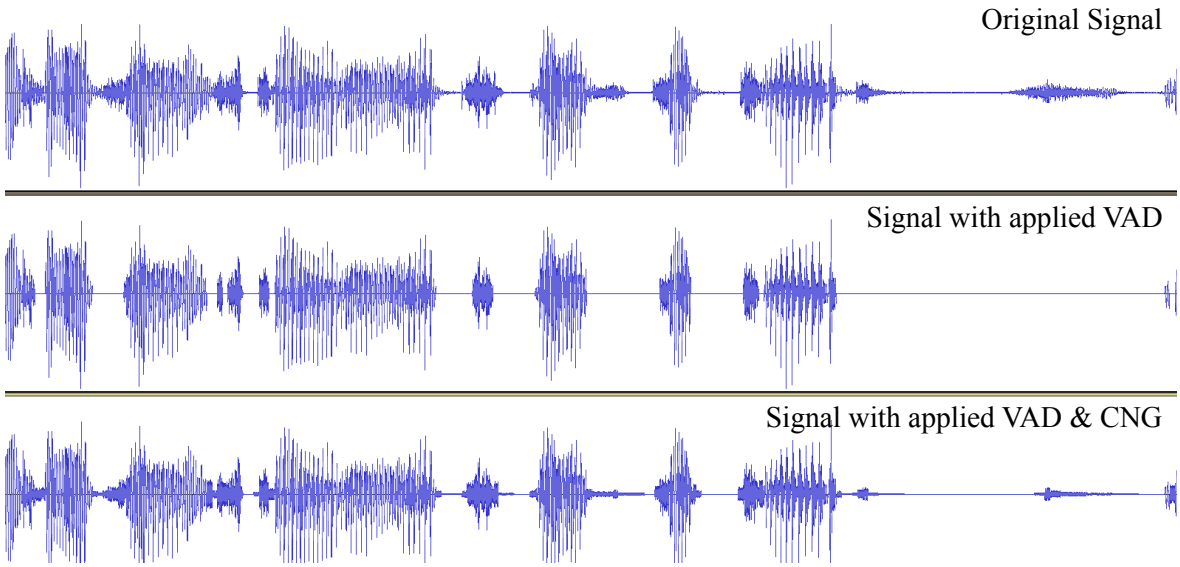


Figure C.2: Waveform effects of VAD and CNG