LinuxSampler Developers
Internet Draft
Document: draft-linuxsampler-protocol-00.txt
Expires: March 2004

C. Schoenebeck
<Affiliation>

Wednesday,
December 31, 2003

# LinuxSampler Control Protocol

## Status of this Memo

This document specifies an application specific protocol for the LinuxSampler core application and arbitrary third party software that interacts with the LinuxSampler application, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. THIS DOCUMENT IS ONLY AN INITIAL DRAFT NOT A FINAL VERSION OF THE PROTOCOL!

## Abstract

The LinuxSampler Control Protocol (LSCP) is an application-level protocol primarily intended for local and remote controlling the LinuxSampler main application, which is a sophisticated console application essentially playing back audio samples and manipulating the samples in real time to certain extent.

## Conventions used in this document

This protocol is always case-sensitive if not explicitly claimed the opposite.

In examples, "C:" and "S:" indicate lines sent by the client and server respectively.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC-2119 [1].

Table of Contents

## 1. Introduction

LinuxSampler is a so called software sampler application capable to playback audio samples from a computer's Random Acess Memory (RAM) as well as directly streaming it from disk. LinuxSampler is designed to be modular. It provides several so called “sampler engines” where each engine is specialized for a certain purpose. LinuxSampler has virtual channels which will be referred in this document as “sampler channels”. The channels are in such way virtual as they can be connected to an arbitrary MIDI input method and arbitrary MIDI channel (e.g. sampler channel 17 could be connected to an ALSA sequencer device 64:0 and listening to MIDI channel 1 there). Each sampler engine will be assigned an own instantance of one of the available sampler engines (e.g. GigEngine, DLSEngine). The audio output of each sampler channel can be routed to an arbitrary audio output method (ALSA / JACK) and an arbitrary audio output channel there.

## 2. Communication Overview

There are two distinct methods of communication between a running instance of LinuxSampler and one or more control applications, so called “frontends”: a simple TCP unidirectional communication method and a TCP / UDP combination for bidirectional communication. The latter needs more effort to be implemented in the frontend application. The two communication methods will be described next.

### 2.1 Simple unidirectional communication

This simple communication method is primarily based on TCP. The frontend application establishes a TCP connection to the LinuxSampler instance on a certain host system. Then the frontend application will send certain ASCII based commands as defined in this document and the LinuxSampler application will response after a certain process time with an appropriate ASCII based answer, also as defined in this document. So this TCP communication is simply based on query and answer paradigm. That way LinuxSampler is only able to answer on queries from frontends, but not able to send messages if it's not asked to. To keep LinuxSampler's informations in the frontend up-tp-date the frontend has to periodically send update commands to get the current informations of the LinuxSampler instance. This is often referred as “polling”. The disadvantage of this simple unidirectional communication approach is obvious: it means network traffic overhead and introduces latency regarding the update of the informations, but is very simple to implement.

## 2.2 Advanced bidirectional communication

This more sophisticated communication method is actually only an extension of the simple unidirectional communication method. The frontend still uses a TCP connection and sends the same commands on the TCP connection, but the frontend has to provide an open UDP port for receiving event messages from the LinuxSampler application. The frontend has to register it's UDP port to the LinuxSampler application by sending the following command on it's TCP connection:

```
SUBSCRIBE NOTIFICATION <udp-port>
```

where <udp-port> will be replaced by the respective UDP port number. If this is accepted by the LinuxSampler application, it will receive events from that point whenever some for the frontend noteworthy event occurred in the LinuxSampler instance. These event UDP packets usually only contain basic informations like the event category and for example on which sampler channel the event occurred. After receiving the event, the frontend has to react by issueing a respective update command on it's TCP connection to get the detail change.

Example: the fill state of a disk stream has changed on sampler channel 3 and the LinuxSampler instance will react by sending the following UDP packet:

```
CHANGE CHANNEL_BUFFER_FILL 3
```

The frontend will react by sending the respictive update command:

```
GET CHANNEL_BUFFER_FILL PERCENTAGE 3
```

to get the fill states of all disk stream buffers on sampler channel 3 and will receive the following answer from LinuxSampler:

```
“[35]62%,[33]80%,[37]98%”
```

Which means there are currently three active streams on sampler channel 3, where the stream with ID “35” is filled by 62%, stream with ID 33 is filled by 80% and stream with ID 37 is filled by 98%.

Beside normal event packets, LinuxSampler will also periodically send PING packets to check if a frontend is still alive. The frontend has to answer with a PONG UDP package (PING and PONG UDP packages will be defined later in this document). If LinuxSampler will not receive such a PONG packet it will consider the frontend to be not available and remove it from the notification list. Such a PING packet is also sent by LinuxSampler when the frontend issued a

"SUBSCRIBE NOTIFICATION" command to check if the given UDP port is really available and not constrained by a firewall for exampler, so the frontend has to open the input UDP port before it tries to register for notification by sending the mentioned command.

# 3. Description for control commands

This chapter will describe the available control commands that can be sent on the TCP connection in detail.

## 3.1 Loading an instrument

An instrument file can be loaded an assigned to a sampler channel by the following command:

```
LOAD INSTRUMENT <filename> <sampler-channel>
```

Where <filename> is the name of the instrument file on the LinuxSampler instance's host system and <sampler-channel> is the number of the sampler channel the instrument should be assigned to. Each sampler channel can only have one instrument.

Possible Answers:

```
"OK" - in case the instrument was successfully loaded
"ERR: <errormesage>" - in case it failed, providing an
                       appropriate error message
```

## 3.2 Loading a sampler engine

A sample engine can be deployed and assigned to a specific sampler channel by the following command:

```
LOAD ENGINE <engine-name> <sampler-channel>
```

Where <engine-name> is usually the C++ class name of the engine implementation and <sampler-channel> the sampler channel the deployed engine should be assigned to. Even if the respective sampler channel has already a engine with that engine name, a new instance will be assinged to the sampler channel.

Possible Answers:

```
"OK" - in case the engine was successfully deployed
"ERR: <errormesage>" - in case it failed, providing an
                       appropriate error message
```

## 3.3 Current number of sampler channels

The number of sampler channels can change on runtime. To get the current amount of sampler channels the frontend can send the the following command:

```
    GET CHANNELS
```

Possible Answers:

    LinuxSampler will answer returning the number of channels.

Example:

```
    C: “GET CHANNELS”
    S: “32”
```

## 3.4 Adding a new sampler channel

A new sampler channel can be added to the end of the sampler channel list by sending the following command:

```
    ADD CHANNEL
```

This will increment the sampler channel count by one and the new sampler channel will appended to the end of the sampler channel list. The frontend will should call the respective calls right after to e.g. load an engine, instrument and setting input, output method on the new channel.

Possible Answers:

```
    “OK[<sampler-channel>]” - in case a new sampler channel could
                              be added, <sampler-channel> reflects
                              the channel number of the new created
                              sampler which should the be used to
                              set up the sampler channel by sending
                              proceeding intialization commands
    “ERR: <errormesage>” - in case it failed, providing an
```

## 3.5 Removing a sampler channel

A sampler channel can be removed by sending the following command:

```
    REMOVE CHANNEL <sampler-channel>
```

This will decrement the sampler channel count by one and also decrement the channel numbers of all subsequent sampler channels by one.

Possible Answers:

```
    “OK” - in case a the sampler channel could be  removed
    “ERR: <errormesage>” - in case it failed, providing an
                           appropriate error message
```

## 3.6 Getting all available engines

The frontend can ask for all available engines by sending the following command:

```
    GET AVAILABLE_ENGINES
```

Possible Answers:

LinuxSampler will answer by sending a comma separated string of the engines' C++ class names.

Example:

```
   C: “GET AVAILABLE_ENGINES”
   S: “GigEngine,AkaiEngine,DLSEngine,JoesCustomEngine”
```

## 3.7 Getting informations about an engine

The frontend can ask for informations about a specific engine by sending the following command:

```
GET ENGINE_INFO <engine-name>
```

Where <engine-name> is usually the C++ class name of the engine implementation.

Possible Answers:

LinuxSampler will answer by sending a <CRLF> separated list string. Each answer line begins with the information category name followed by a colon and then a space character <SP> and finally the info string to that info category. At the moment the following categories are defined:

```
DESCRIPTION – arbitrary description text about the engine
VERSION     - arbitrary version string regarding engine
              version
```

The mentioned fields above don't have to be in particular order.

Example:

```
C: “GET ENGINE_INFO JoesCustomEngine”
S: “DESCRIPTION: this is Joe's custom sampler engine”
   “VERSION: testing-1.0”
```

## 3.8 Getting sampler channel informations

The frontend can ask for the current settings of a sampler channel by sending the following command:

```
GET CHANNEL_INFO <sampler-channel>
```

Where <sample-channel> is the sampler channel the frontend is interested in.

Possible Answers:

LinuxSampler will answer by sending a <CRLF> separated list string. Each answer line begins with the settings category name followed by a colon and then a space character <SP> and finally the info string to that setting category. At the moment the following categories are defined:

```
ENGINE_NAME – name of the engine that is deployed on the
```

```
                  sampler channel, “<NONE>” if there's no engine
                  deployed yet for this sampler channel
    AUDIO_OUTPUT_TYPE – output system which used to output the
                        audio signal (at the moment either
                        “ALSA” or “JACK”)
    AUDIO_OUTPUT_CHANNEL – the physical output channel for the
                           audio signal
    INSTRUMENT – the file name of the loaded instrument,
                 “<NONE>” if there's no instrument yet loaded
                 for this sampler channel
    MIDI_INPUT_TYPE – at the moment only “ALSA”
    MIDI_INPUT_PORT – string representing the input MIDI port
                      (in case of ALSA e.g. “64:0”)
    MIDI_INPUT_CHANNEL – the MIDI input channel number this
                         sampler channel should listen to
    VOLUME – channel volume factor (where a value < 1.0 means
             attenuation and a value > 1.0 means amplification)
```

The mentioned fields above don't have to be in a particular order.

Example:

```
C: “GET CHANNEL_INFO 34”
S: “ENGINE_NAME: GigEngine”
   “VOLUME: 1.0”
   “AUDIO_OUTPUT_TYPE: ALSA”
   “AUDIO_OUTPUT_CHANNEL: 8”
   “INSTRUMENT: /home/joe/FazioliPiano.gig”
   “MIDI_INPUT_TYPE: ALSA”
   “MIDI_INPUT_PORT: 64:0“
   “MIDI_INPUT_CHANNEL: 5”
```

## 3.9 Current number of active voices

The frontend can ask for the current number of active voices on a sampler channel by sending the following command:

```
GET CHANNEL_VOICE_COUNT <sampler-channel>
```

Where <sampler-channel> is the sampler channel the frontend is interested in.

Possible Answers:

LinuxSampler will answer by returning the number of active voices on that channel.

## 3.10 Current number of active disk streams

The frontend can ask for the current number of active disk streams on a sampler channel by sending the following command:

```
GET CHANNEL_STREAM_COUNT <sampler-channel>
```

Where <sampler-channel> is the sampler channel the frontend is interested in.

Possible Answers:

LinuxSampler will answer by returning the number of active disk streams on that channel in case the engine supports disk streaming, if the engine doesn't support disk streaming it will return “NA” for not available.

## 3.11 Current fill state of disk stream buffers

The frontend can ask for the current fill state of all disk streams on a sampler channel by sending the following command:

```
GET CHANNEL_BUFFER_FILL BYTES <sampler-channel>
```

to get the fill state in bytes or

```
GET CHANNEL_BUFFER_FILL PERCENTAGE <sampler-channel>
```

to get the fill state in percent, where <sampler-channel> is the sampler channel the frontend is interested in.

Where <sampler-channel> is the sampler channel the frontend is interested in.

Possible Answers:

LinuxSampler will answer by returning a comma separated string with the fill state of all disk stream buffers on that channel, or “NA” for not available in case the engine doesn't support disk streaming. Each entry in the answer list will begin with the stream's ID followed by a space character <SP> and the numerical representation of the fill size (either in bytes or percentage).

Example:

```
C: “GET CHANNEL_BUFFER_FILL BYTES”
S: “[115] 420500, [116] 510300, [75] 110000, [120] 230700”
```

## 3.12 Setting audio output type

The frontend can alter the audio output type on a specific sampler channel by sending the following command:

```
SET CHANNEL AUDIO_OUTPUT_TYPE <audio-output-type> <sampler-channel>
```

Where <audio-output-type> is currently either “ALSA” or “JACK” and <sampler-channel> is the respective sampler channel.

Possible Answers:

```
“OK” - on success
“ERR: <errormesage>” - in case it failed, providing an
                       appropriate error message
```

## 3.13 Setting audio output channel

The frontend can alter the audio output channel on a specific sampler channel by sending the following command:

```
SET CHANNEL AUDIO_OUTPUT_CHANNEL <audio-channel> <sampler-channel>
```

Where <audio-channel> is the physical output channel where the audio signal of this sampler channel should be routed to and <sampler-channel> is sampler channel where this should happen.

Possible Answers:

```
“OK” - on success
“ERR: <errormesage>” - in case it failed, providing an
                       appropriate error message
```

## 3.14 Setting MIDI input port

The frontend can alter the input MIDI port on a specific sampler channel by sending the following command:

```
    SET CHANNEL MIDI_INPUT_PORT <midi-input-port> <sampler-channel>
```

Where <midi-input-port> is the MIDI input port string (in case of ALSA for example “64:0” and <sampler-channel> is the sampler channel where this should be set.

Possible Answers:

```
    “OK” - on success
    “ERR: <errormesage>” - in case it failed, providing an
                           appropriate error message
```

## 3.15 Setting MIDI input channel

The frontend can alter the MIDI channel a sampler channel should listen to by sending the following command:

```
SET CHANNEL MIDI_INPUT_CHANNEL <midi-input-chan> <sampler-channel>
```

Where <midi-input-chan> is the new MIDI input channel where <sampler-channel> should listen to.

Possible Answers:

```
    “OK” - on success
    “ERR: <errormesage>” - in case it failed, providing an
                           appropriate error message
```

## 3.16 Setting channel volume

The frontend can alter the volume of a sampler channel by sending the following command:

```
SET CHANNEL VOLUME <volume> <sampler-channel>
```

Where <volume> is an optionally dotted positive number (a value smaller than 1.0 means attenutation and a value greater than 1.0 means amplification) and <sampler-channel> defines the sampler channel where this volume factor should be set.

Possible Answers:

```
“OK” - on success
“ERR: <errormesage>” - in case it failed, providing an
                       appropriate error message
```

## 3.17 Registering frontend for receiving UDP event messages

The frontend can register itself to the LinuxSampler application to be informed about noteworthy events by sending this command:

```
SUBSCRIBE NOTIFICATION <udp-port>
```

Where <udp-port> is the UDP port number on the frontend's host on which the frontend will listen to. The frontend has to open, listen and react on that port before it tries to register itself for NOTIFICATION, because the LinuxSampler instance will send a PING packet to test if the UDP is actually reachable and the frontend is listening on that port. The frontend will then immediately have to answer by sending a PONG packet, else the SUBSCRIBE NOTIFICATION command will fail (see UDP chapter for PING and PONG packets). The LinuxSampler instance will periodically send PING packets on which the frontend has to answer, else LinuxSampler assumes the frontend to be not available and will stop to send notification / event messages.

Possible Answers:

```
“OK” - on success
“ERR: <errormesage>” - in case it failed, providing an
                       appropriate error message
```

## 3.18 Deregister frontend for not receiving UDP event messagesanymore

The frontend can deregister itself if doesn't want to receive UDP event packets by sending the following command:

```
UNSUBSCRIBE NOTIFICATION [<udp-port>]
```

Where <udp-port> is the UDP port number on which the frontend recently received event packets, if not provided it will deregister all registered ports on the frontend's host system.

Possible Answers:

```
“OK” - on success
“ERR: <errormesage>” - in case it failed, providing an
                       appropriate error message
```

## 4. Command Syntax

The following are the LSCP (LinuxSampler control protocol) commands:

```
  ADD <SP> CHANNEL

  GET <SP> <get-instruction>

  LOAD <SP> <load-instruction>

  REMOVE <SP> CHANNEL <SP> <sampler-channel>

  SET <SP> CHANNEL <SP> <set-chan-instruction> <SP> <sampler-channel>

  SUBSCRIBE <SP> NOTIFICATION <SP> <udp-port>

  UNSUBSCRIBE <SP> NOTIFICATION [<SP> <udp-port>]
```

The syntax of the above argument fields is given below using Backus-Naur Form (BNF as described in RFC-2234 [2]) where applicable.

```
<get-instruction> ::= AVAILABLE_ENGINES |
                      CHANNELS |
                      CHANNEL_INFO <SP> <sampler-channel> |
                      CHANNEL_BUFFER_FILL <SP> <buffer-size-type> |
                      CHANNEL_STREAM_COUNT |
                      CHANNEL_VOICE_COUNT |
                      ENGINE_INFO <SP> <engine-name>

<load-instruction> ::= INSTRUMENT <SP> <load-instr-args> |
                       ENGINE <SP> <load-engine-args>

<sampler-channel> ::= <number>

<set-chan-instruction> ::=
                AUDIO_OUTPUT_CHANNEL <audio-output-channel> |
                AUDIO_OUTPUT_TYPE <audio-output-type> |
                MIDI_INPUT_PORT <midi-input-port> |
                MIDI_INPUT_CHANNEL <midi-input-channel> |
                MIDI_INPUT_TYPE <midi-input-type> |
                VOLUME <dotnum>

<udp-port> ::= <number>

<buffer-size-type> ::= BYTES | PERCENTAGE

<engine-name> ::= <cpp-classname>

<load-instr-args> ::= <filename> <SP> <sampler-channel>
```

```
<load-engine-args> ::= <engine-name> <SP> <sampler-channel>

<audio-output-channel> ::= <number>

<audio-output-type> ::= ALSA | JACK

<midi-input-port> ::= <string>

<midi-input-channel> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
                         11 | 12 | 13| 14 | 15 | 16

<midi-input-type> ::= ALSA

<cpp-classname> ::= class name as defined by the C++ programming
                    language

<filename> ::= <string>

<string> ::= <char> | <char> <string>

<char> ::= <c> | "\" <x>

<c> ::= any one of the 128 ASCII characters, but not any
                      <special> or <SP>

<special> ::= "<" | ">" | ";" | ":" | "&" | "{" | "}" | the control
              characters (ASCII codes 0 through 31 inclusive and 127)

<dotnum> ::= <snum> "." <number>

<number> ::= <d> | <d> <number>

<d> ::= any one of the ten digits 0 through 9

<snum> ::= arbitrary number of digits representing a decimal
           integer value in the range including 0 to infinity

<CRLF> ::= <CR> <LF>

<CR> ::= the carriage return character (ASCII code 13)

<LF> ::= the line feed character (ASCII code 10)

<SP> ::= the space character (ASCII code 32)

<x> ::= any one of the 128 ASCII characters (no exceptions)
```

# 5. Events and special UDP packets

This chapter will describe all currently defined UDP packets sent by LinuxSampler.

## 5.1 Number of sampler channels changed

In this case LinuxSampler will send the following packet:

```
“CHANGE CHANNELS”
```

## 5.2 Number of active voices changed

In this case LinuxSampler will send a packet with following shape:

```
CHANGE CHANNEL_VOICE_COUNT <sampler-channel>
```

Where <sampler-channel> will be replaced by the sampler channel the voice count change occurred.

## 5.3 Number of active disk streams changed

In this case LinuxSampler will send a packet with following shape:

```
CHANGE CHANNEL_STREAM_COUNT <sampler-channel>
```

Where <sampler-channel> will be replaced by the sampler channel the stream count change occurred.

## 5.4 Disk stream buffer fill state changed

In this case LinuxSampler will send a packet with following shape:

```
CHANGE CHANNEL_BUFFER_FILL <sampler-channel>
```

Where <sampler-channel> will be replaced by the sampler channel the buffer fill state change occurred.

## 5.5 Channel informations changed

In this case LinuxSampler will send a packet with following shape:

```
    CHANGE CHANNEL_INFO <sampler-channel>
```

Where <sampler-channel> will be replaced by the sampler channel the channel info change occurred.

## 5.6 Special packet PING

Sense behind this packet is to check if the frontend is (still) listening on it's registered UDP port. This special packet has this shape:

```
    PING <udp-port> <string>
```

Where <string> is an arbitrary character string that has to be returned by sending a PONG UDP packet to the UDP port given by <udp-port>.

## 5.7 Special packet PONG

This packet has to be returned by the frontend in reaction to a PING packet received from LinuxSampler and a PONG packet looks like this:

```
    PONG <string>
```

Where <string> is a character string transmited with PING, which should be send in order to confirm the PING packet.

Example:

```
    S: “PING 2067 ahj_89zdiQ”
    C: “PONG ahj_89zdiQ”      (sent to port 2067 of LinuxSampler's
                               host)
```

## 6. Event Syntax

The following are the defined event messages sent via UDP (only in case the frontend registered itself to receive UDP event packets):

```
  CHANGE <SP> <event-type>

  PING <SP> <udp-port> <SP> <string>

  PONG <SP> <string>
```

The syntax of the above argument fields is given below using Backus-Naur Form (BNF as described in RFC-2234 [3]) where applicable.

```
    <event-type> ::= CHANNELS |
                     CHANNEL_VOICE_COUNT <sampler-channel> |
                     CHANNEL_STREAM_COUNT <sampler-channel> |
                     CHANNEL_BUFFER_FILL <sampler-channel> |
                     CHANNEL_INFO <sampler-channel>

    <udp-port> ::= <number>

    <sampler-channel> ::= <number>

    <string> ::= <char> | <char> <string>

    <char> ::= <c> | "\" <x>

    <c> ::= any one of the 128 ASCII characters, but not any
            <special> or <SP>

    <special> ::= "<" | ">" | ";" | ":" | "&" | "{" | "}" | the
                  control characters (ASCII codes 0 through 31
                  inclusive and 127)

    <number> ::= <d> | <d> <number>

    <d> ::= any one of the ten digits 0 through 9

    <x> ::= any one of the 128 ASCII characters (no exceptions)

    <SP> ::= the space character (ASCII code 32)
```

## Security Considerations

As there is so far no method of authentication and authorisation defined and so not required for a client applications to succeed to connect, running LinuxSampler might be a security risk for the host system the LinuxSampler instance is running on.

## References

< Your references will be listed here. View "Page Layout" if they are not currently visible. >

## Acknowledgments

<Add any acknowledgements>

## Author's Addresses

<Firstname> <Lastname>
<Affiliation>
<Address>
Phone: <optional>
Email: <Your email address>

1 Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997

2 Crocker, D. and Overell, P.(Editors), "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, Internet Mail Consortium and Demon Internet Ltd., November 1997

3 Crocker, D. and Overell, P.(Editors), "Augmented BNF for Syntax Specifications: ABNF", RFC 2234, Internet Mail Consortium and Demon Internet Ltd., November 1997