Py-Raptor TV Server

A prototype for
streaming DVB-T television

Laurent Fasnacht, Amin Shokrollahi, Giovanni Cangiani

Outline

- why
- what
- who
- how
- TODO

Why

- an example of real-world application to promote Raptor (the best FEC code ever invented) and our lab (ALGO);
- TCP is not good for video streaming over long distance and it does not scale. UDP alone is not good either because video becomes unwatchable at < 0.5% loss;
- Raptor allows to have many concurrent sources for the same stream
- Amin wants to watch TV (football) while abroad;

Target scenarios

- global internet television: watch your favorite tv shows from anywhere in the world (e.g. slingbox)

- keep the video watchable in lossy network
- optimize bandwidth usage
  - ➡ adapt stream bitrate
  - ➡ make best use of available bandwidth
  - ➡ good quality at any distance from the source

- broadcast live television on a controlled network (e.g. bluewin-tv)

- scale for thousands of users
  - ➡ avoid tcp and unicast. Use multicast instead
  - ➡ protect from packet loss
- fast channel change

Video Streaming

comparison chart

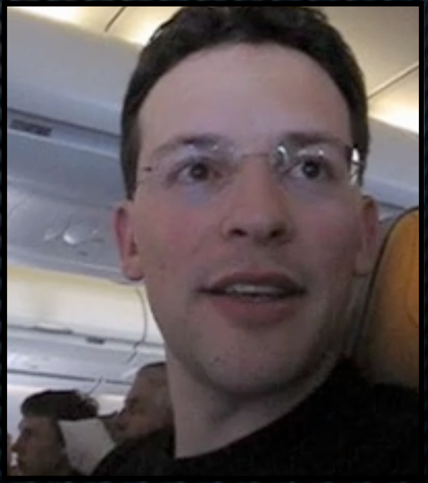
	TCP	UDP	Raptor
Best video quality for given bandwidth	Only on short RTT	Only on non lossy network (~nowhere)	Yes
No glitches	Yes	Only on non lossy network (~nowhere)	Yes
Scale using mcast	No	Yes	Yes
Easy multiple source	No	No	Yes

Our prototype system

- input from DVB-T signal from cable TV;
- unicast streaming for users @ home and on EPFL WiFi (our slingbox);
- multicast streaming within EPFL local network (our bluewin-tv);
- Stream using Raptor protected UDP;
- Many channels and users in parallel.

WHO

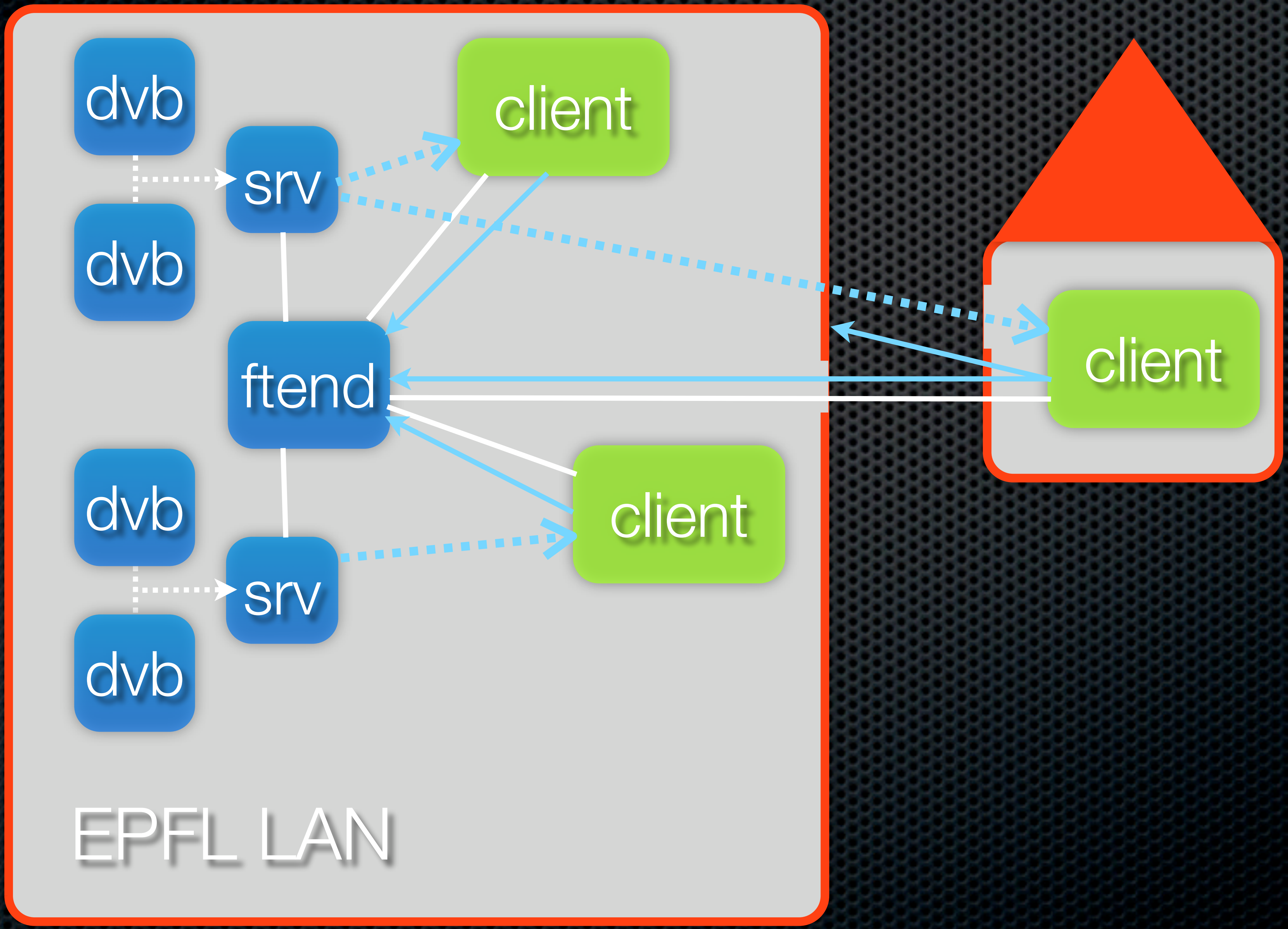
- Zeno Crivelli
- Laurent Fasnacht
- Nicolas Heiniger
- Damir Laurenzi
- Amin Shokrollahi



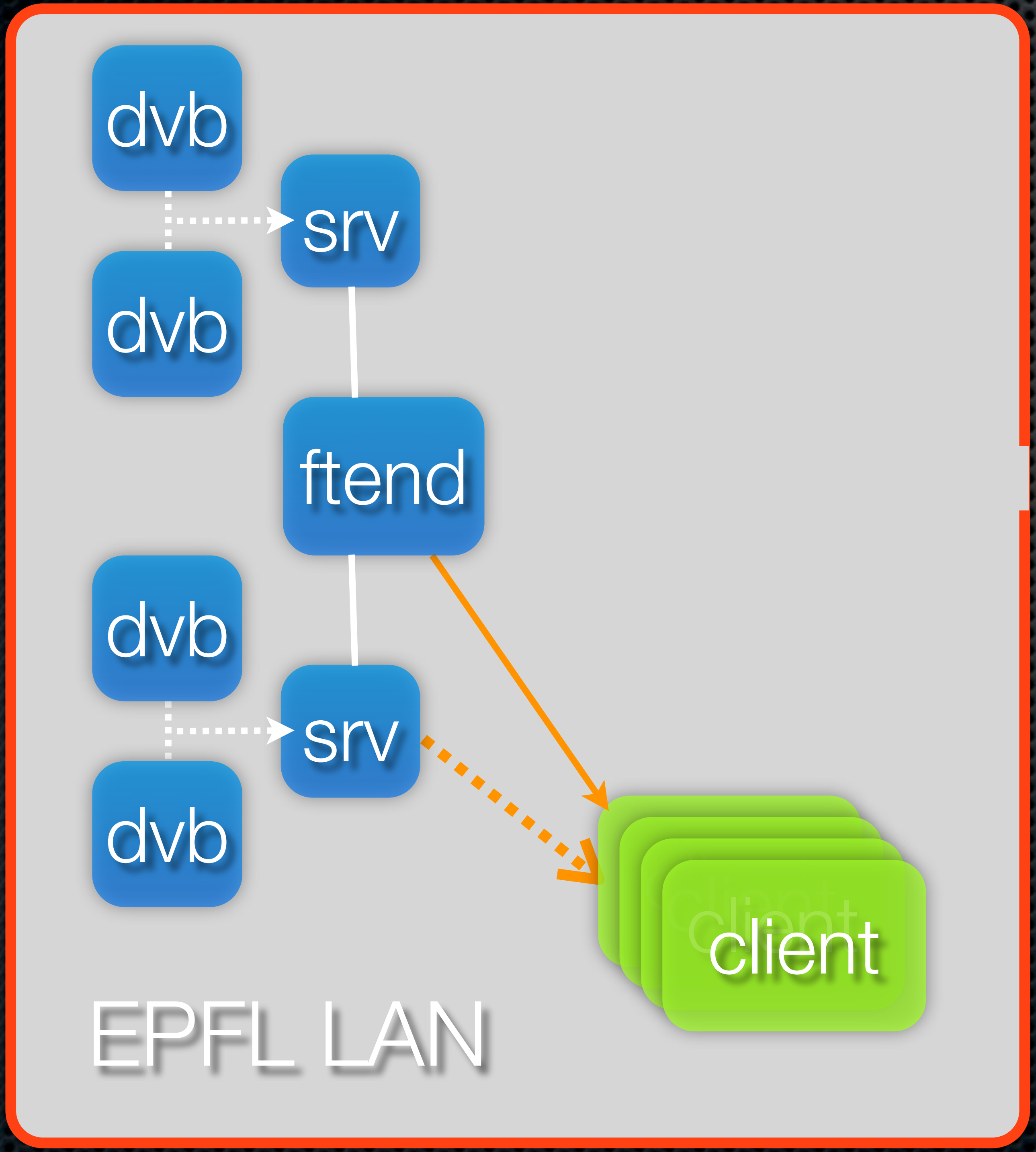
Laurent Fasnacht

- new server code
- prototype client
- linux sys optimization
- hardware test/selection

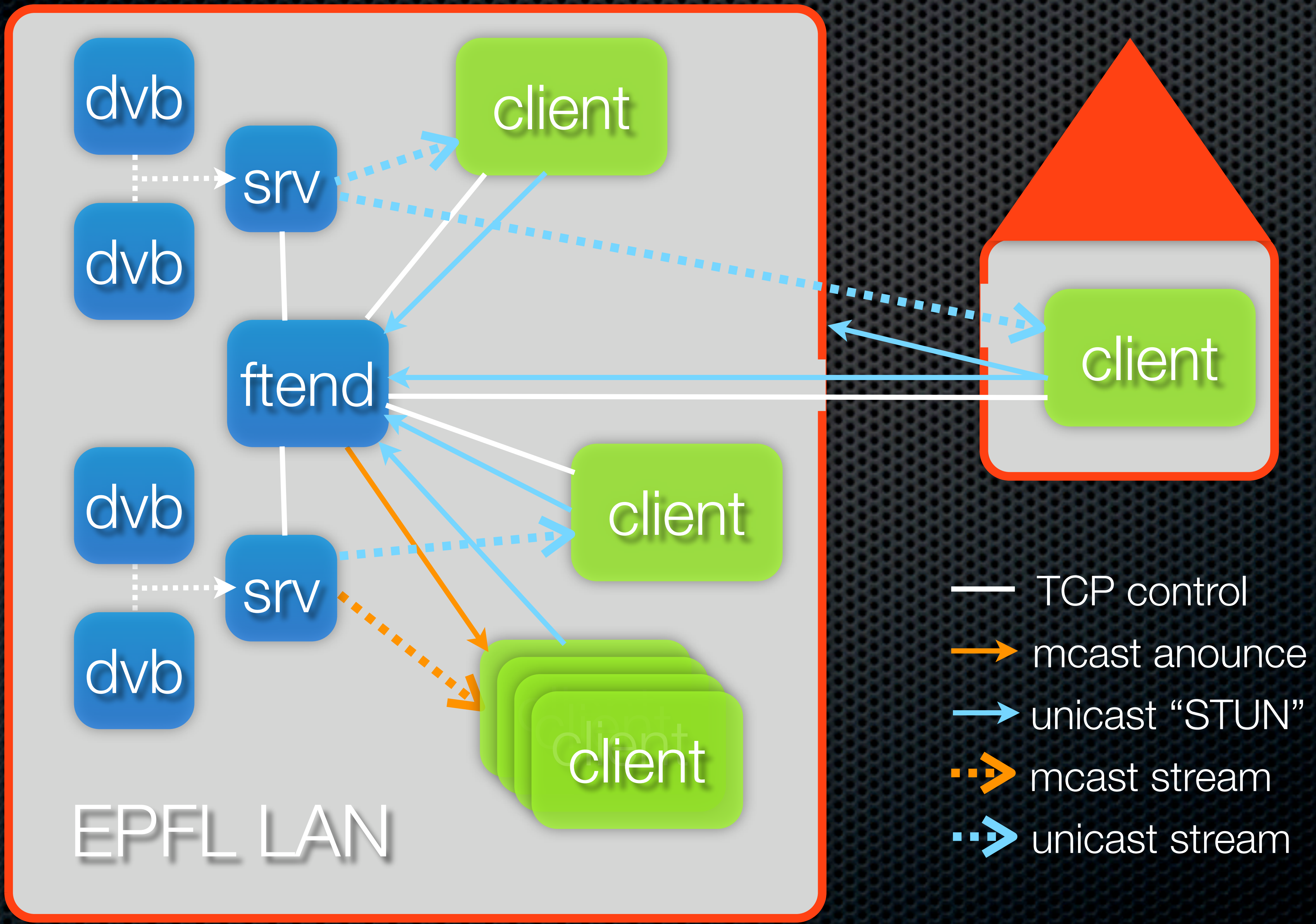
System Overview (1)



System Overview (2)



System Overview



— TCP control

→ mcast anounce

→ unicast “STUN”

⇢ mcast stream

⇢ unicast stream

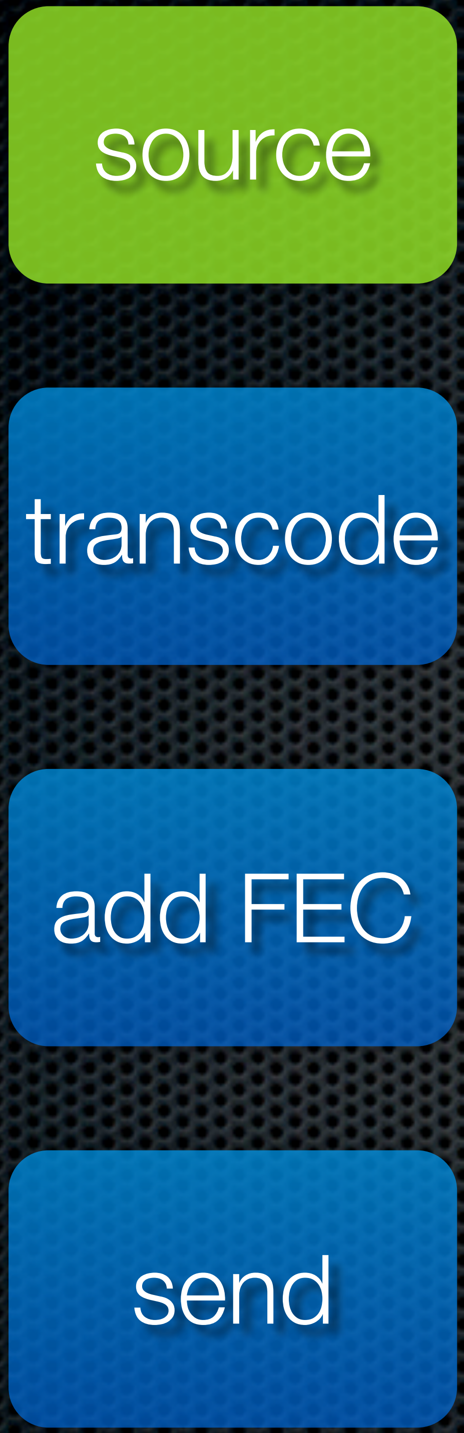
How - outline

- front-end
- server concept
- few implementation details

Front-end (proxy)

- collect information on available channels from servers
- advertise multicast channels
- listen for TCP connection from [unicast] clients and sends the xml list of channels/options
- process client’s requests: check availability, select good server, reply to the client and forward the request to the designated server
- proxy for STUN/activity messages
- stop unused streams

Stream Server Concept



source

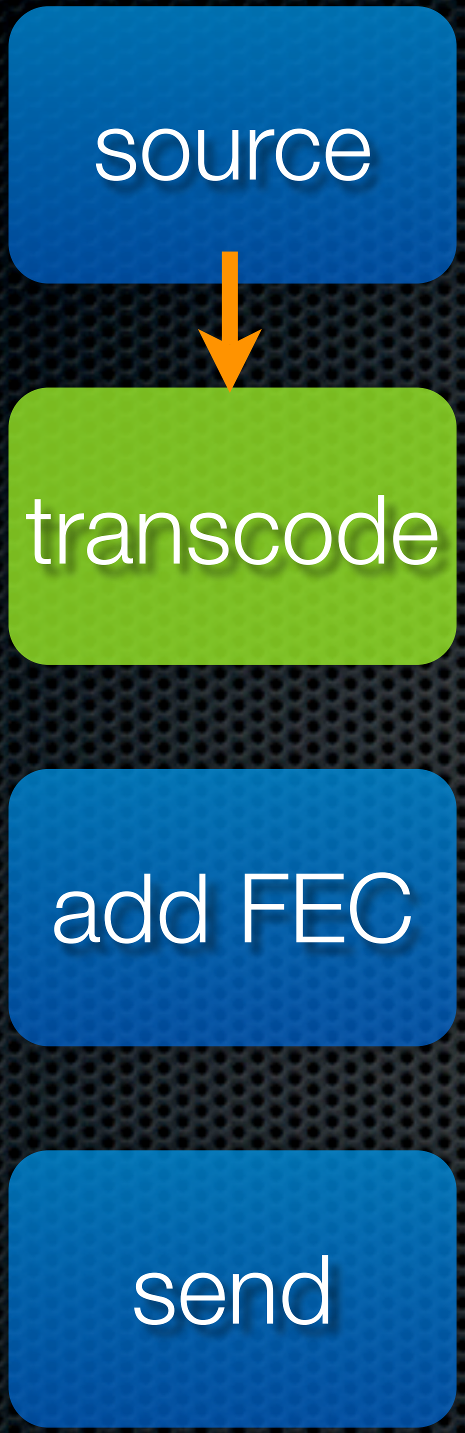
transcode

add FEC

send

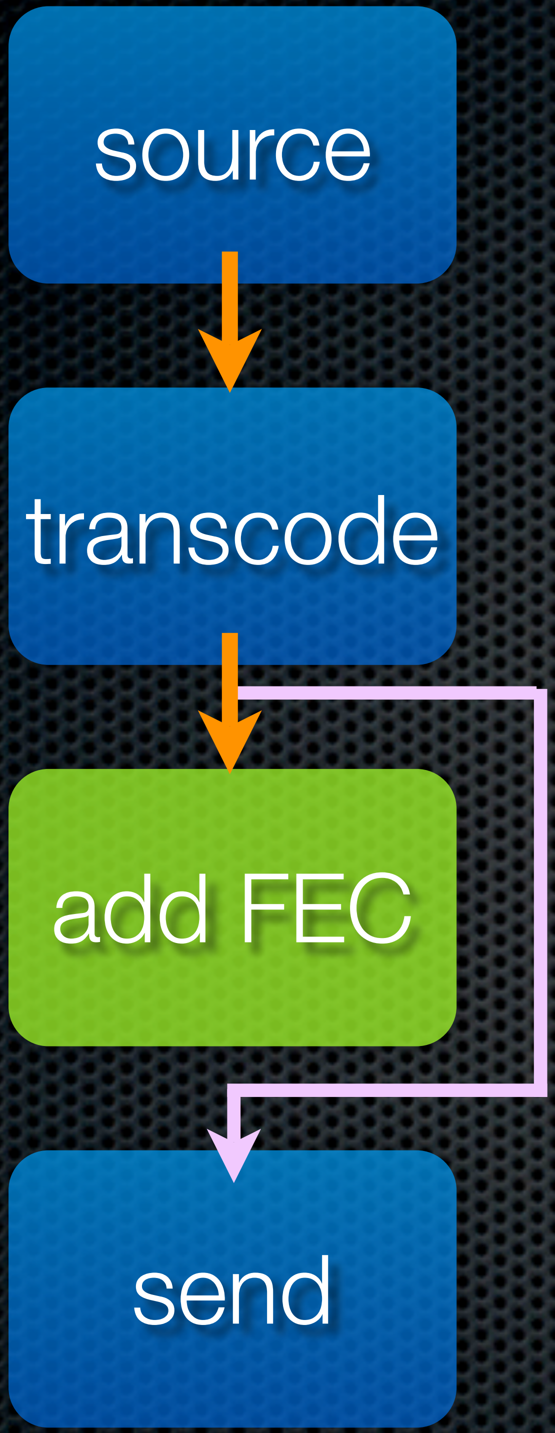
- single MPEG2-TS stream
- 4 ÷ 8 Mbit/s
- 188 Bytes/packet
  including 32bit header
- bursty

Stream Server Concept



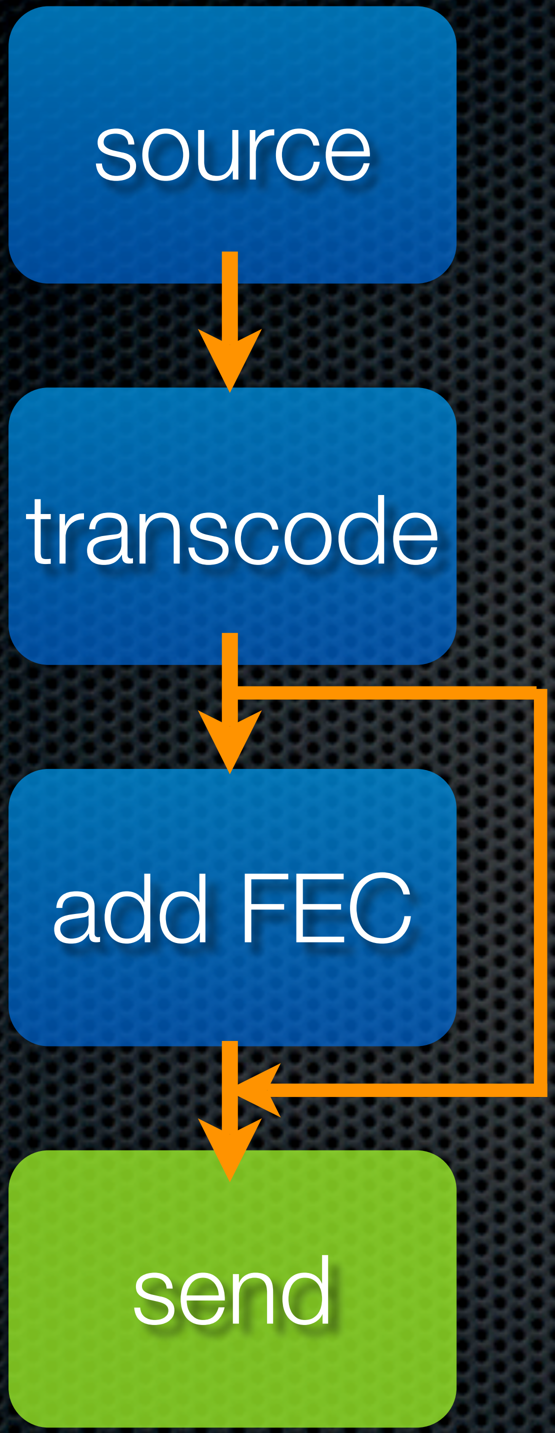
- ffmpeg
- MPEG4 or H264
- optimized for various bit-rates 700÷4000 Kbit/s
- problem: adds >2 sec of delay

Stream Server Concept



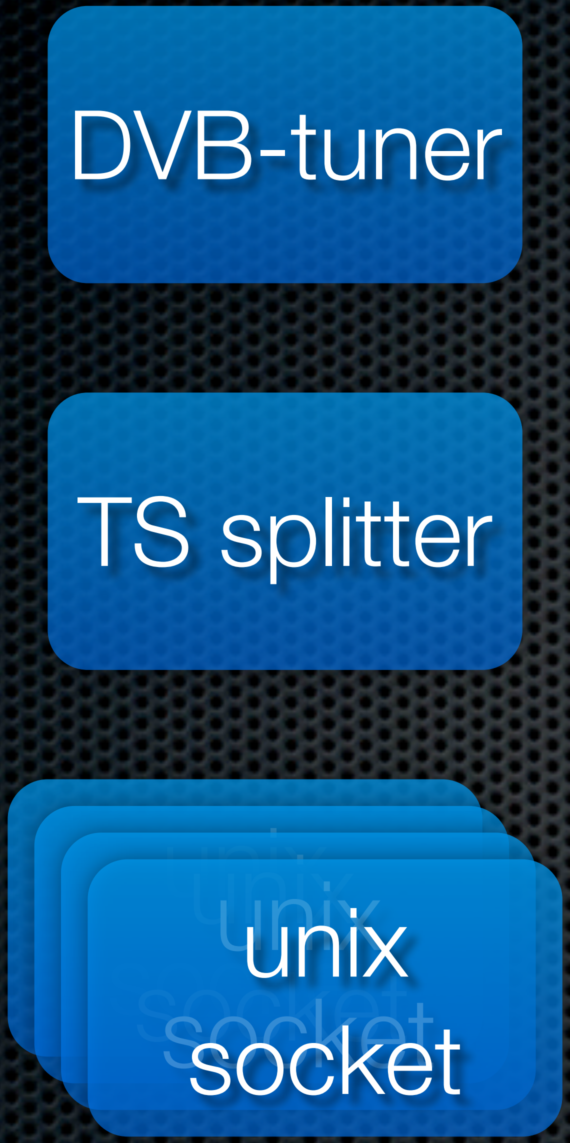
- systematic Raptor (in progress)
- live => small blocks (source ~128 pkts)
- live => fixed no. repair symbols
- RS for comparison (in progress)

Stream Server Concept



- 2÷5 blocks concurrently sent
- better if packets are uniformly distributed over time
- ready packets are stored (interleaved) on the same buffer
- 1320 Bytes/packet (7*188+4)
- multicast or [many] unicast

Source: DVB-T feed details

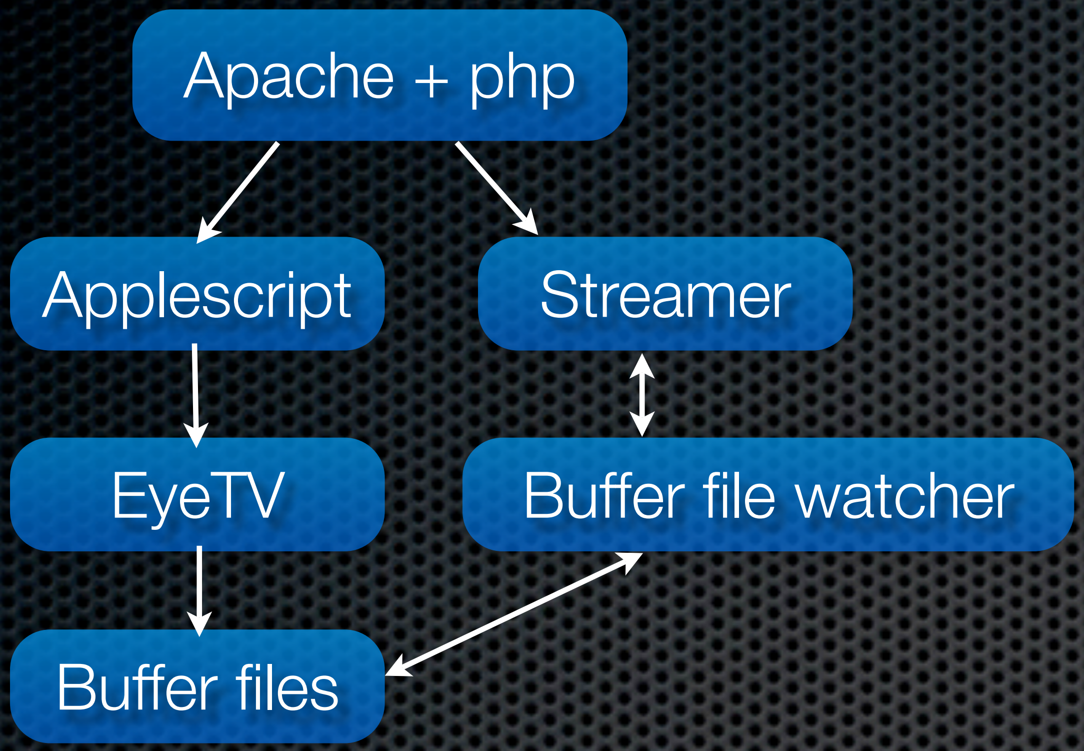

DVB-tuner

TS splitter

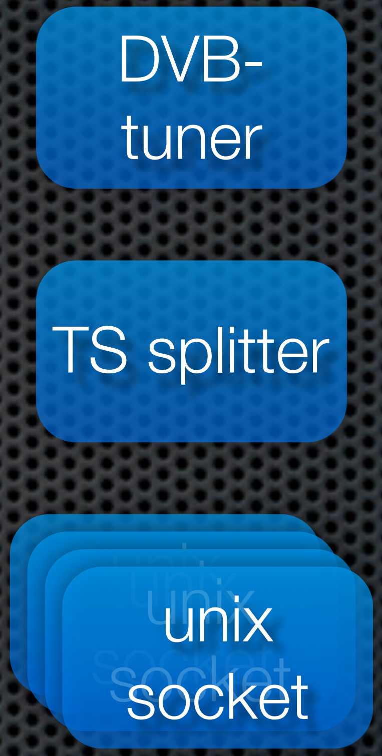
unix socket

- DVB-tuners (2 per server, Linux driver)
- fix frequency
- up to 8 channels per frequency
- channels are composed of various packet streams (audio, video, subtitles...) all with the same PID.
- discard PAT packets
- split channels (send to distinct unix socket)

Much simpler than version 0



vs.



- many external programs (also closed source)
- single channel
- useless hard disk usage
- huge delay

- only kernel and a C program of 100 lines
- multiple channels
- very small cpu usage
- no delay

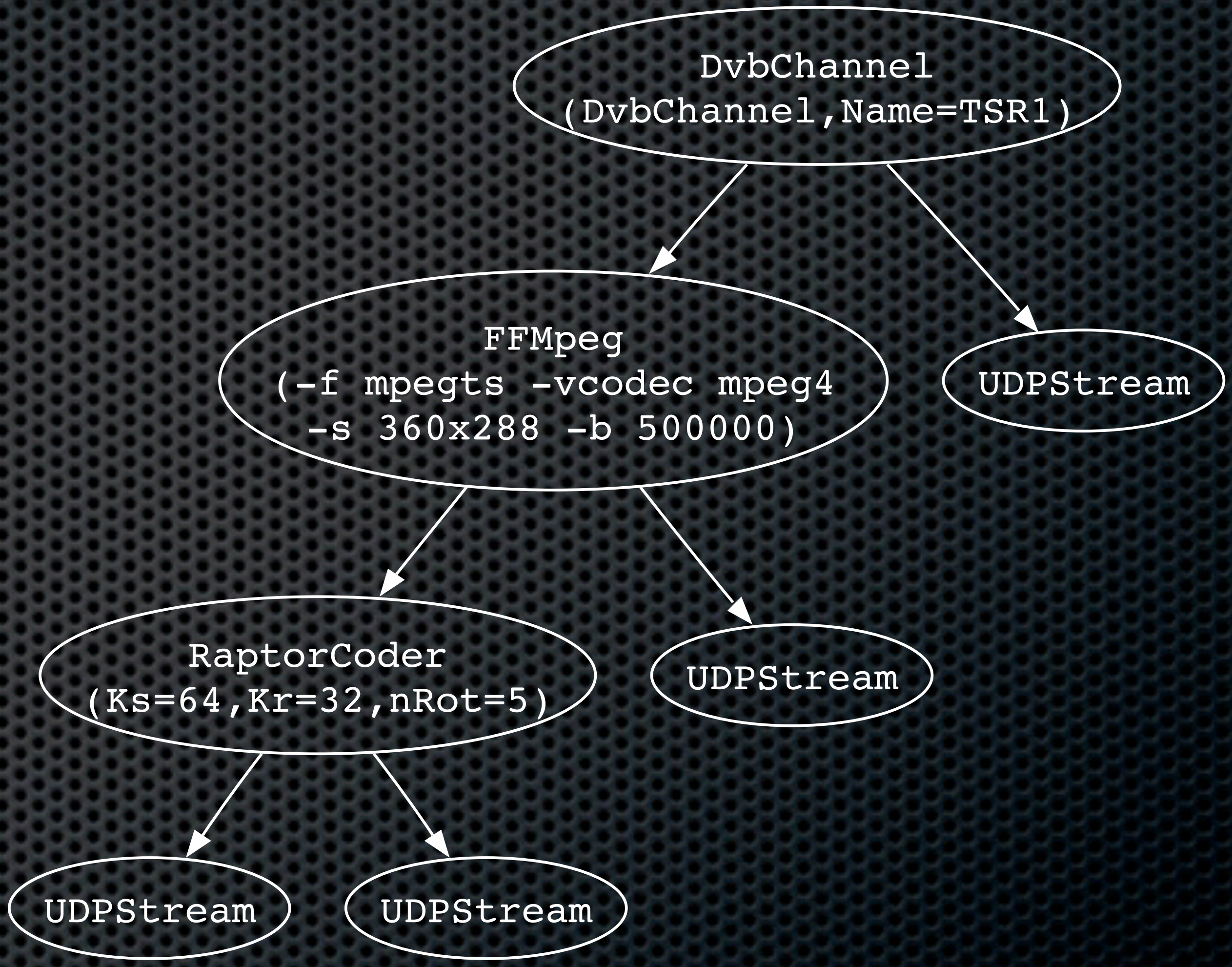
Server Implementation Details

- written in python => pragmatic, nice, didactic
- includes an user friendly Python interface to DFRaptor
- extensible modular architecture
- configured using a simple channel description xml file

Server Implementation Details

Streaming Chains

- every stream is a chain of modules auto-generated from a unique program description string
- the server recycles as many modules as possible on multiple requests



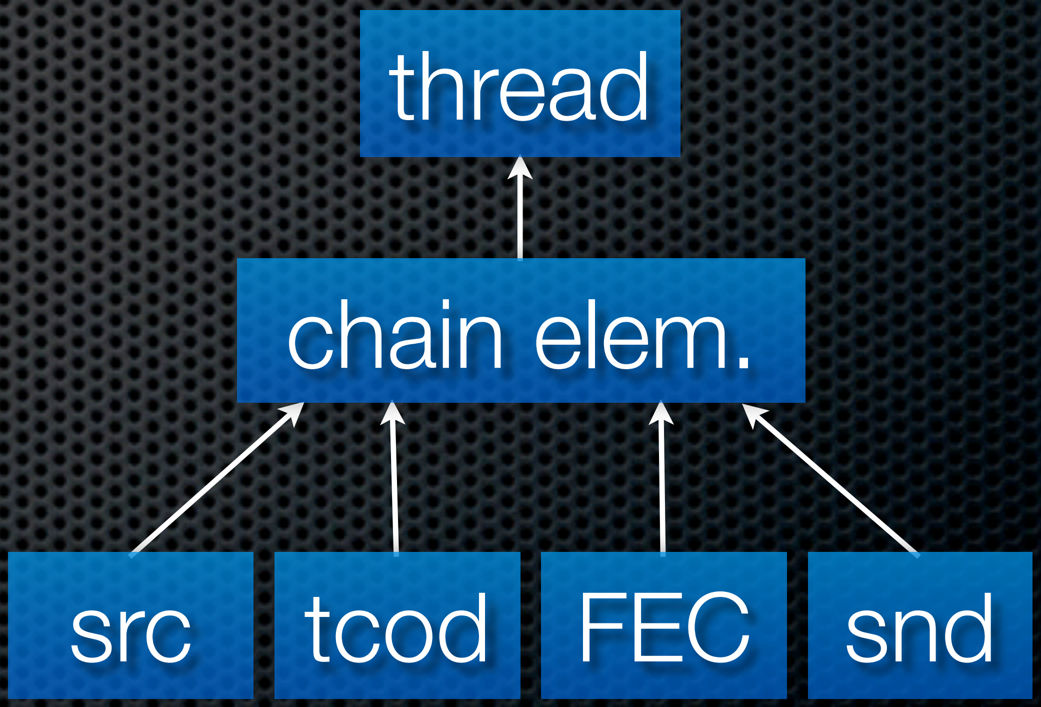
program description examples:

```
Dvb,Name=TSR1|RaptorCoder,Ks=64,Kr=32,nRot=5
Dvb,Name=TSR1|FFMpeg,params=-f mpegts -vcodec mpeg4 -s 360x288 -b 500000|
RaptorCoder,Ks=128,Kr=32,nRot=5
```

Server Implementation Details

Server modules

- get input from parent
- process
- store on output buffer
- manage access to buffer from children



thread

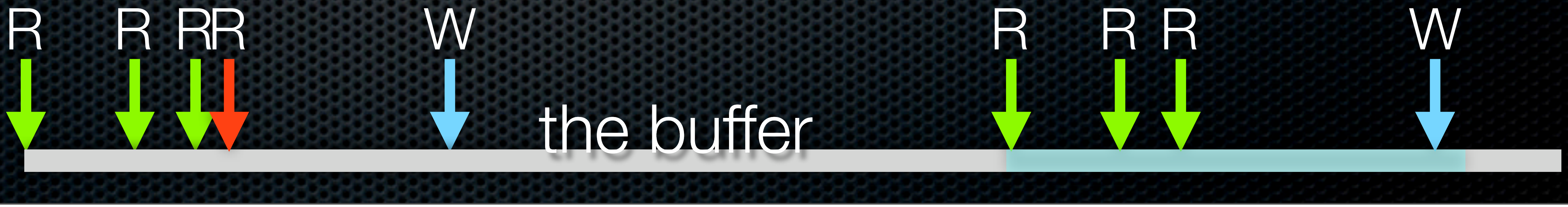
chain elem.

src

tcod

FEC

snd



R R RR W R R R W

the buffer

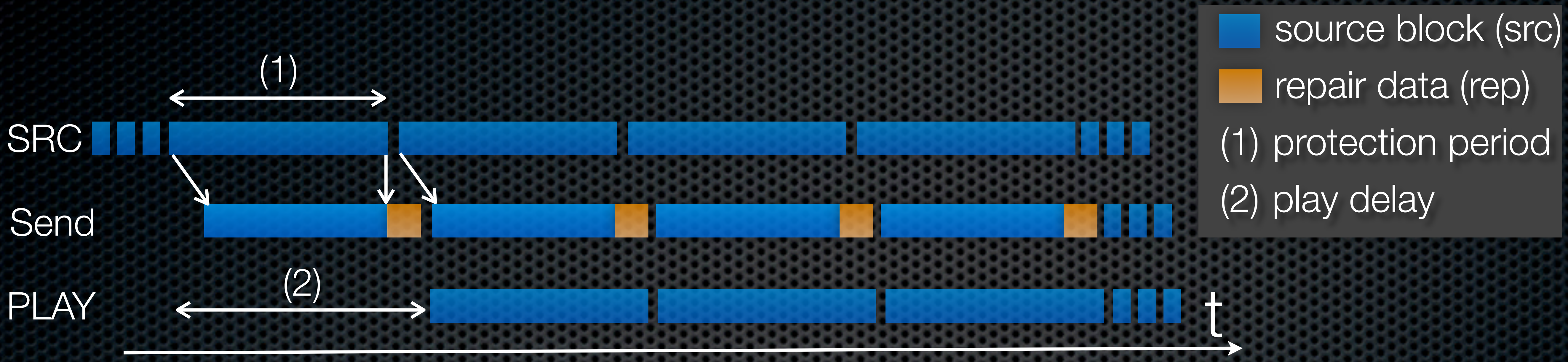
The big enemy: delay

- small delay = faster startup time
- short delay = faster channel change
- a lot done but we can do better

Sources of delay:

- protocol < 0.5 s (unicast only)
- transcoding~1÷2 s (not relevant in channel change)
- not using systematic ~ 0.3 s (may be)
- player buffer > 3 s

Live TV



- Live => repair is finite and fixed (~ rep < 0.3 * src)
- play delay = src + rep [+ extra player buffer]
- send bitrate = src bitrate * (rep+src)/src $\gtrsim$ client bandwidth
- smaller protection period (src) => shorter play delay
- larger protection period => less sensible to burst loss

Recordings (vod) extension

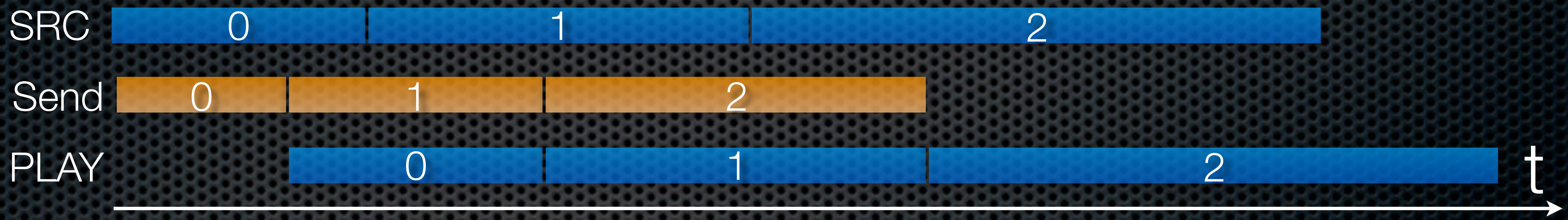
- Multiple senders

Fountain

- Protect against any loss

any Z is ok

- Nicer buffering and block partitioning schemes:



SRC 0 1 2

Send 0 1 2

PLAY 0 1 2

t

- Better video codec (e.g. 2 pass h264)
- P2P

TODO (0)

- built-in player (in progress)
  - ➡ embeddable
  - ➡ better control of player buffer
  - ➡ smart block size
  - ➡ drastically reduce delay
- send source as soon as we get it (in progress)

TODO (1) “easy”

- GUI client
- try alternative buffer in chain element
- optimize parameters (block size, protection, block interleaving) for various network condition and video bit-rates
- let the client tune the amount of protection by sending repair packets on different channels
- built-in h264 transcoder, progressive stream

TODO (2) “harder”

- embedded linux client (prototype set-top-box)
- fluendo elisa plugin (=> gstreamer module)
- distributed collaborative slingbox