Tiny functions for lots of things

Keith Winstein

joint work with: Francis Y. Yan [S], Sadjad Fouladi [S], John Emmons [S], Riad S. Wahby [S], Emre Orbay [S], Brennan Shacklett [S], William Zeng [S], Dan Iter [S], Shuvo Chaterjee, Catherine Wu [SHS]
Daniel Reiter Horn [D], Ken Elkabany [D], Chris Lesniewski-Laas [D],
Karthikeyan Vasuki Balasubramaniam [UCSD], Rahul Bhalerao [UCSD], George Porter [UCSD], Anirudh Sivaraman [MIT]

[S] Stanford University
[SHS] Saratoga High School
[D] Dropbox
[UCSD] UC San Diego
[MIT] MIT

Message of this talk

- A little "functional-ish" programming goes a long way.
- It's worth refactoring megamodules (codecs, TCP, compilers, machine learning) using ideas from functional programming.
- Just the ability to **name, save, and restore** program states is powerful in its own right.

Breaking megamodules into functions

Lepton: JPEG recompression in a distributed filesystem

ExCamera: Fast interactive video encoding

Salsify: Videoconferencing with co-designed codec and transport protocol

gg: IR for “laptop to lambda” jobs with 8,000-way parallelism

Breaking megamodules into functions

Lepton: JPEG recompression in a distributed filesystem

- "functional" JPEG codec for boundary-oblivious **sharding**

ExCamera: Fast interactive video encoding

- "functional" video codec for fine-grained **parallelism**

Salsify: Videoconferencing with co-designed codec and transport protocol

- "functional" codec to **explore an execution path** without committing

gg: IR for "laptop to lambda" jobs with 8,000-way parallelism

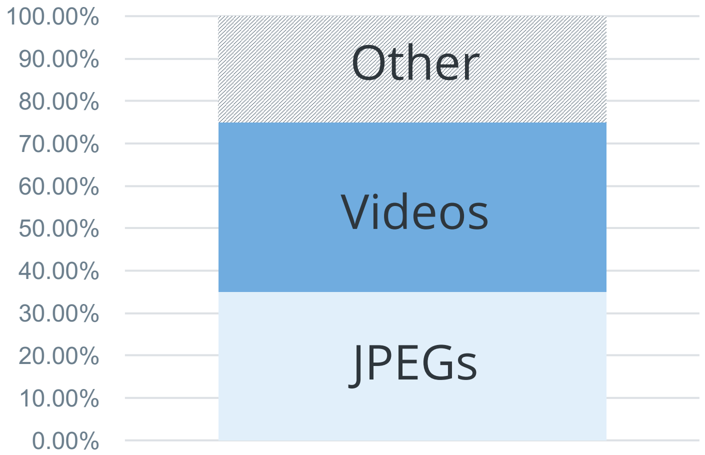
- "functional" representation of **practical parallel pipelines**

System 1: Lepton (distributed JPEG recompression)

Daniel Reiter Horn, Ken Elkabany, Chris Lesniewski-Lass, and KW, **The Design, Implementation, and Deployment of a System to Transparently Compress Hundreds of Petabytes of Image Files for a File-Storage Service**, in NSDI 2017 (Community Award winner).

Storage Overview at Dropbox

- ¾ Media



- Roughly an Exabyte in storage
- Can we save backend space?

JPEG File

- Header
- 8x8 blocks of pixels
  - DCT transformed into 64 coefs
    - Lossless
  - Each divided by large quantizer
    - Lossy
  - Serialized using Huffman code
    - Lossless

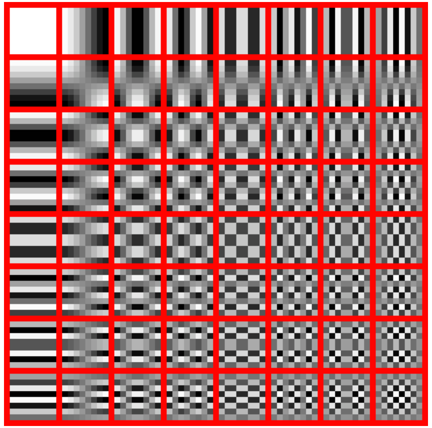
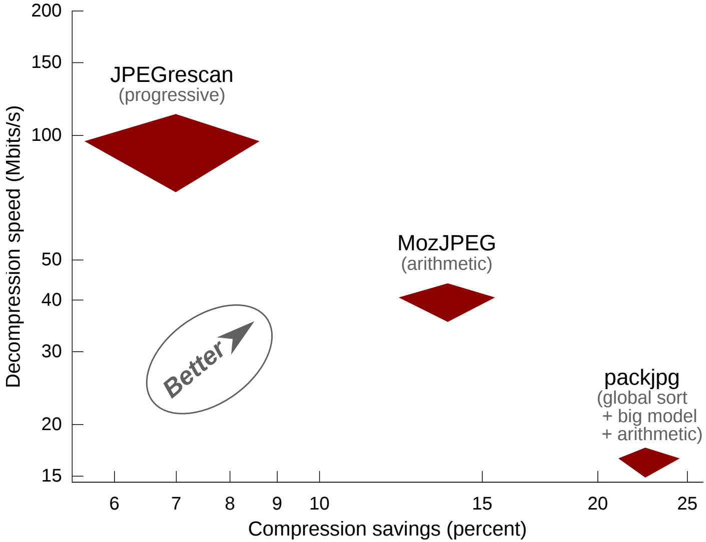


Image credit: wikimedia

Idea: save storage with transparent recompression

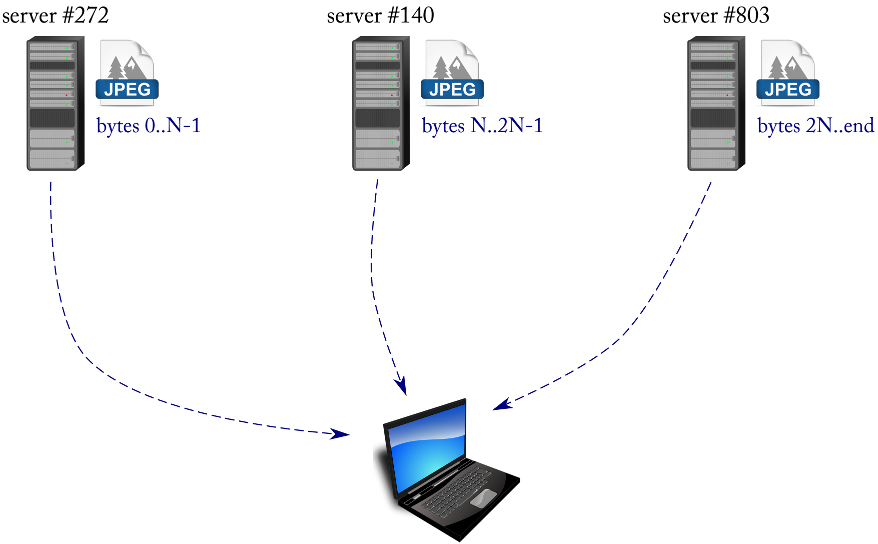
- **Requirement:** byte-for-byte reconstruction of original file
- **Approach:** improve bottom "lossless" layer only
- Replace DC-predicted Huffman code with an arithmetic code
- Use a probability model to predict "1" vs. "0"

Prior work

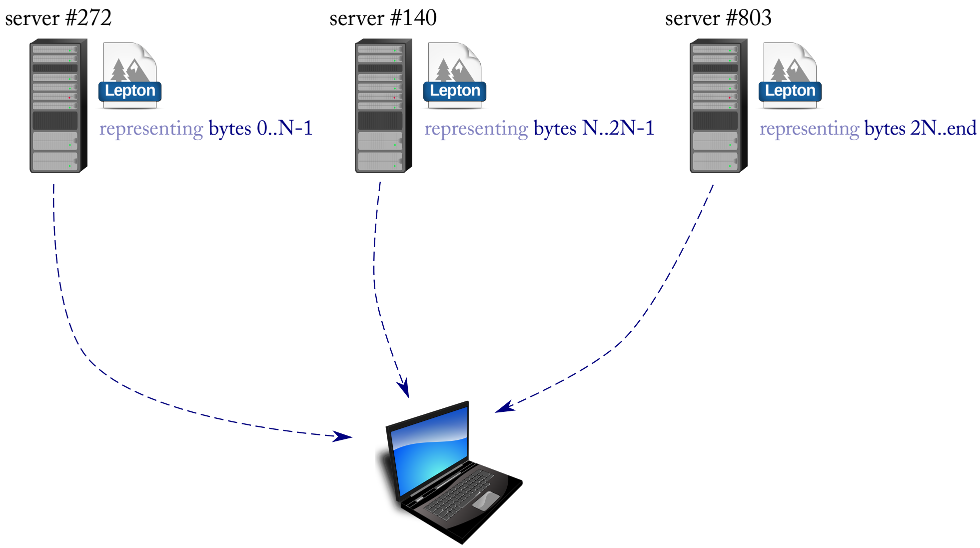


Decompression speed (Mbits/s)
200
150
100
50
40
30
20
15
6
7
8
9
10
15
20
25
Compression savings (percent)
JPEGrescan
(progressive)
MozJPEG
(arithmetic)
packjpg
(global sort
+ big model
+ arithmetic)
Better

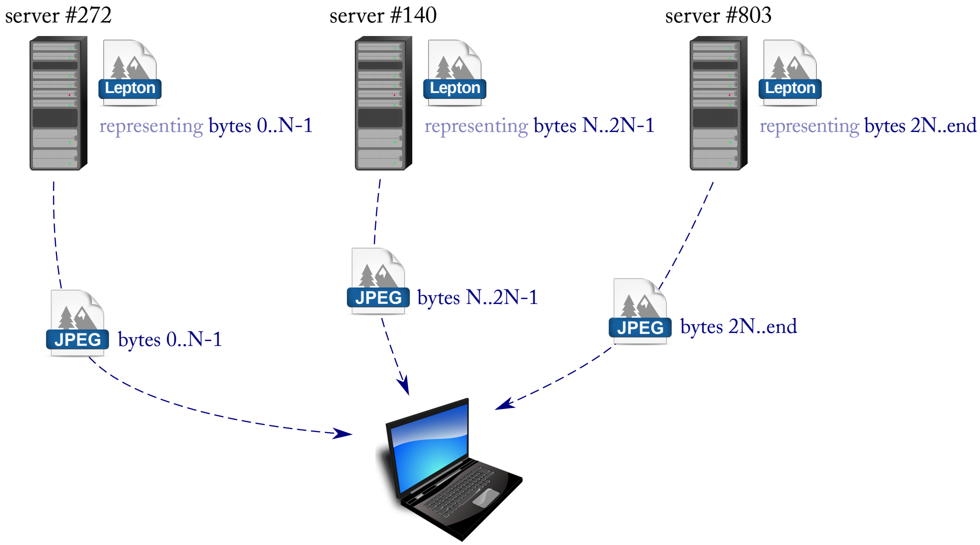
Challenge: distributed filesystem with arbitrary chunk boundaries



Challenge: distributed filesystem with arbitrary chunk boundaries



Challenge: distributed filesystem with arbitrary chunk boundaries



Requirements for distributed compression

- Store and decode file in independent chunks
  - Can start at any byte offset
- Achieve $>$ 100 Mbps decoding speed per chunk
- Don't lose data
  - Immune to adversarial/pathological input files
  - Every time program changed, qualify on a billion images
  - Three compilers (with and without sanitizers) must match on all billion images

Challenges

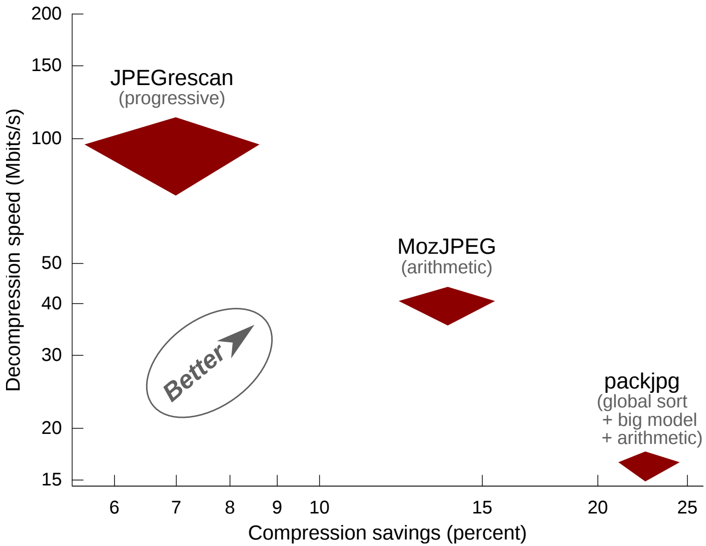
- Baseline JPEG is encoded as a *stream* of Huffman codewords with opaque state (DC prediction).
- `encode(HuffmanTable, vector<Coefficient>)` $\rightarrow$ `vector<bit>`
- How to encode chunk of original file, starting in midstream?
  - Midstream = in the middle of a Huffman codeword
  - Midstream = unknown DC (average) value

When the client retrieves a chunk of a JPEG file, how does the fileserver re-encode that chunk **from** Lepton **back to** JPEG?

Making the state of the JPEG encoder explicit

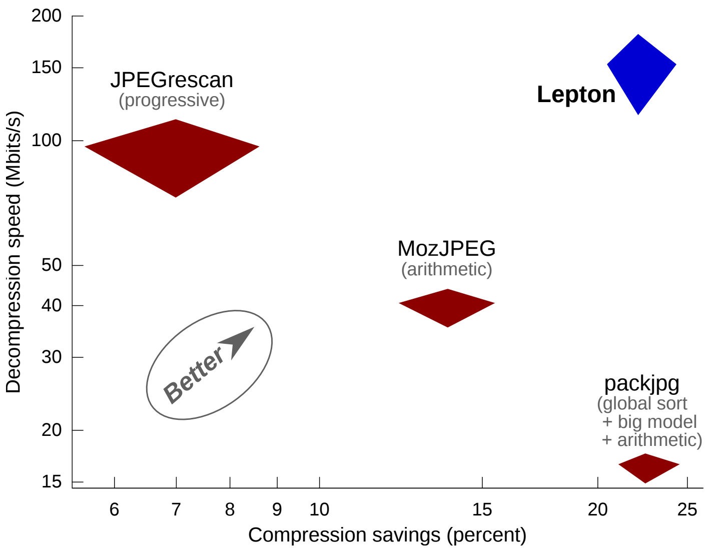
- Formulate JPEG encoder in **explicit state-passing style**
- Implement DC-predicted Huffman encoder that can resume from any byte boundary
- ```
  encode(HuffmanTable, vector<bit>, int dc, vector<Coefficient>)
  → vector<bit>
  ```
  ```

Results



200
150
100
50
40
30
20
15
Decompression speed (Mbits/s)
JPEGrescan
(progressive)
MozJPEG
(arithmetic)
packjpg
(global sort
+ big model
+ arithmetic)
Better
6
7
8
9
10
15
20
25
Compression savings (percent)

Results

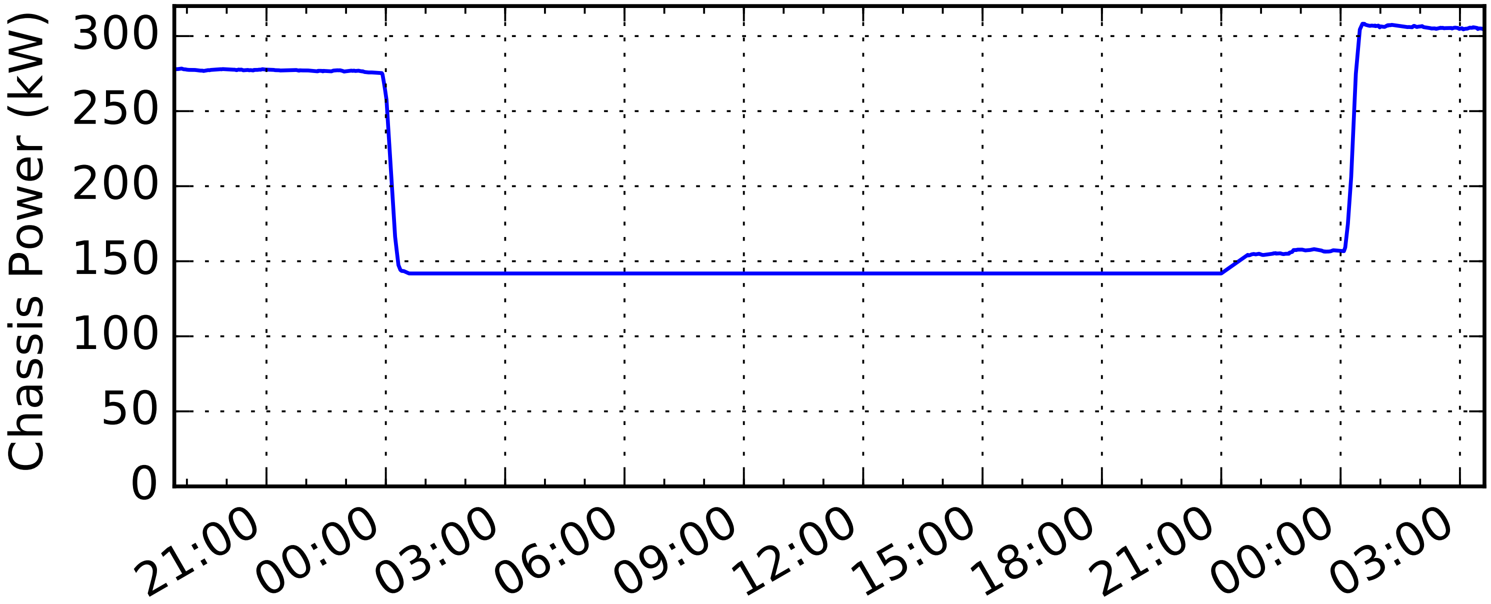


200
150
100
50
40
30
20
15
Decompression speed (Mbits/s)
6
7
8
9
10
15
20
25
Compression savings (percent)
JPEGrescan
(progressive)
Lepton
MozJPEG
(arithmetic)
packjpg
(global sort
+ big model
+ arithmetic)
Better

Deployment

- Lepton has encoded 150 billion files
  - 203 PiB of JPEG files
  - Saving 46 PiB
  - So far…
    - Backfilling at > 6000 images per second

Power Usage at 6,000 Encodes



Lepton concluding thoughts

- A little bit of functional programming can go a long way.
- Functional JPEG codec lets Lepton **distribute** decoding with arbitrary chunk boundaries and **parallelize** within each chunk.

System 2: ExCamera (fine-grained parallel video processing)

Sadjad Fouladi, Riad S. Wahby, Brennan Shacklett, Karthikeyan Vasuki Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and KW, **Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads**, in NSDI 2017.

https://ex.camera

What we currently have

Google Docs

- People can make changes to a word-processing document
- The changes are instantly visible for the others

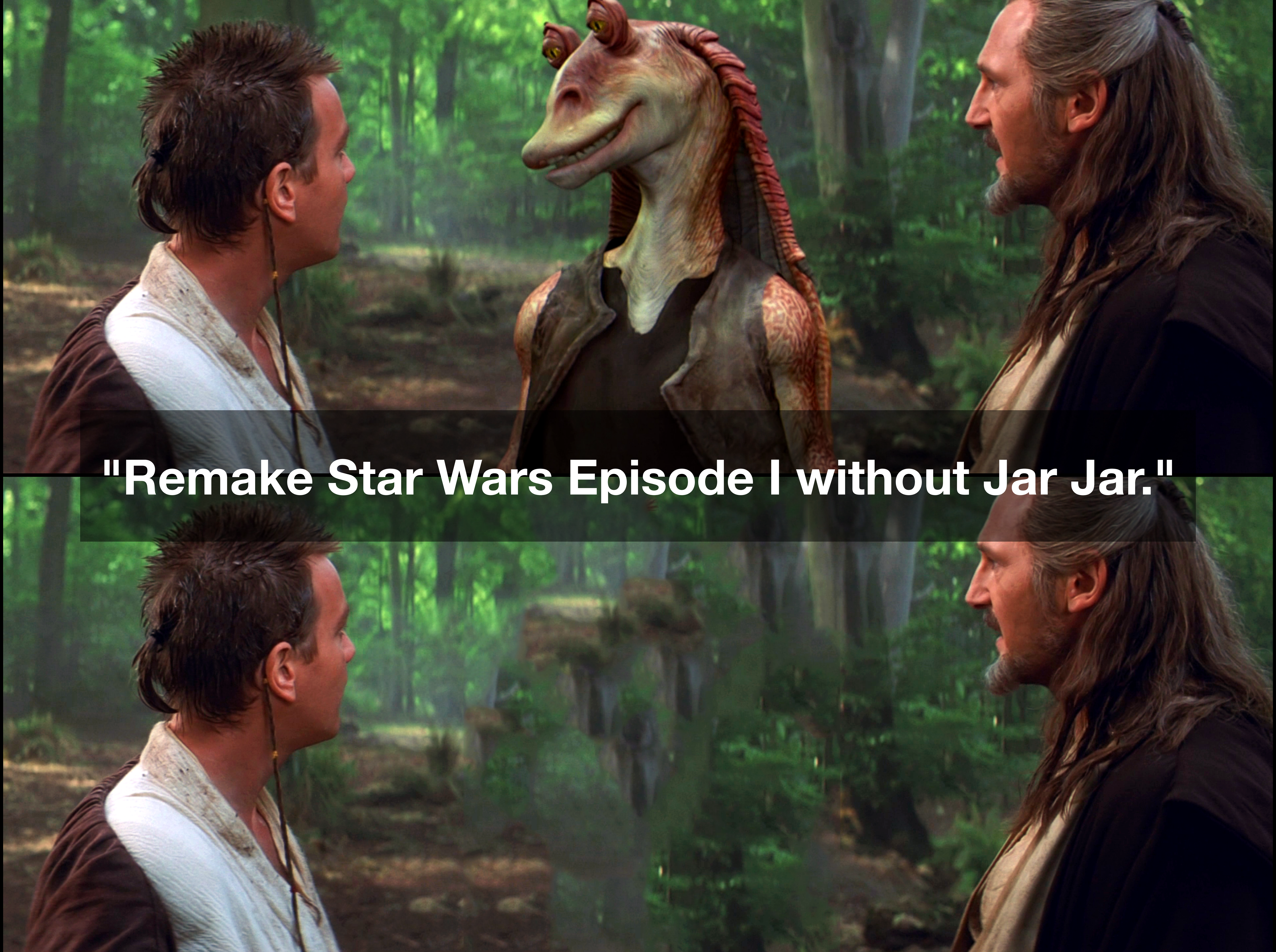
What we would like to have

Google Docs *for Video*?

- People can interactively edit and transform a video
- The changes are instantly visible for the others


"Apply this awesome filter to my video."


"Look everywhere for this face in this movie."


"Remake Star Wars Episode I without Jar Jar."

The Problem

Currently, running such pipelines on videos takes hours and hours, even for a short video.

The Question

Can we achieve interactive collaborative video editing by using massive parallelism?

The challenges

- Low-latency video processing would need **thousands of threads**, **running in parallel**, with **instant startup.**
- However, **the finer-grained the parallelism, the worse the compression efficiency.**

Enter *ExCamera*

- We made two contributions:
  - Framework to run **5,000-way parallel jobs** with IPC on a commercial “cloud function” service.
  - Purely functional video codec for **massive fine-grained parallelism**.
- We call the whole system **ExCamera**.

Cloud function services have (as yet) unrealized power

- AWS Lambda, Google Cloud Functions
- Intended for event handlers and Web microservices, *but...*
- Features:
  - ✔ Thousands of threads
  - ✔ Arbitrary Linux executables
  - ✔ Sub-second startup
  - ✔ Sub-second billing

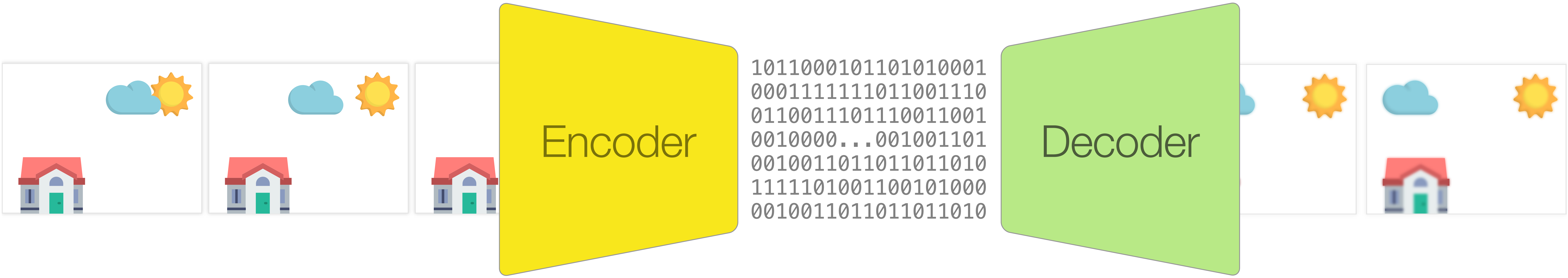
3,600 threads for one second → 9¢

Now we have the threads, but...

- With the existing encoders, the finer-grained the parallelism, the worse the compression efficiency.

Video Codec

- A piece of software or hardware that compresses and decompresses digital video.



How video compression works

- Exploit the temporal redundancy in adjacent images.
- Store the first image on its entirety: a **key frame**.
- For other images, only store a "diff" with the previous images: an **interframe**.

In a 4K video @15Mbps, a key frame is **~1 MB**, but an interframe is **~25 KB**.

Existing video codecs only expose a simple interface

compressed video

```
encode([🌄,🌄,...,🌄]) → keyframe + interframe[2:n]

decode(keyframe + interframe[2:n]) → [🌄,🌄,...,🌄]
```

Traditional parallel video encoding is limited

serial ↓

encode(i[1:200]) → keyframe_1 + interframe[2:200]

parallel ↓

[thread 01] encode(i[1:10]) → kf_1 + if[2:10]

[thread 02] encode(i[11:20]) → kf_{11} + if[12:20] **+1 MB**

[thread 03] encode(i[21:30]) → kf_{21} + if[22:30] **+1 MB**

⋮

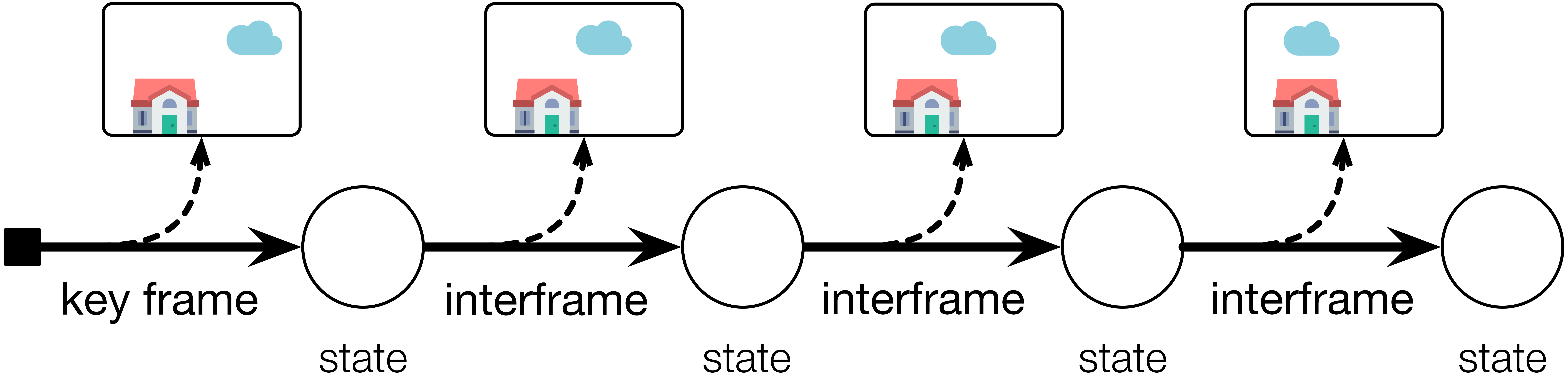
[thread 20] encode(i[191:200]) → kf_{191} + if[192:200] **+1 MB**

finer-grained parallelism ⇒ more key frames ⇒ worse compression efficiency

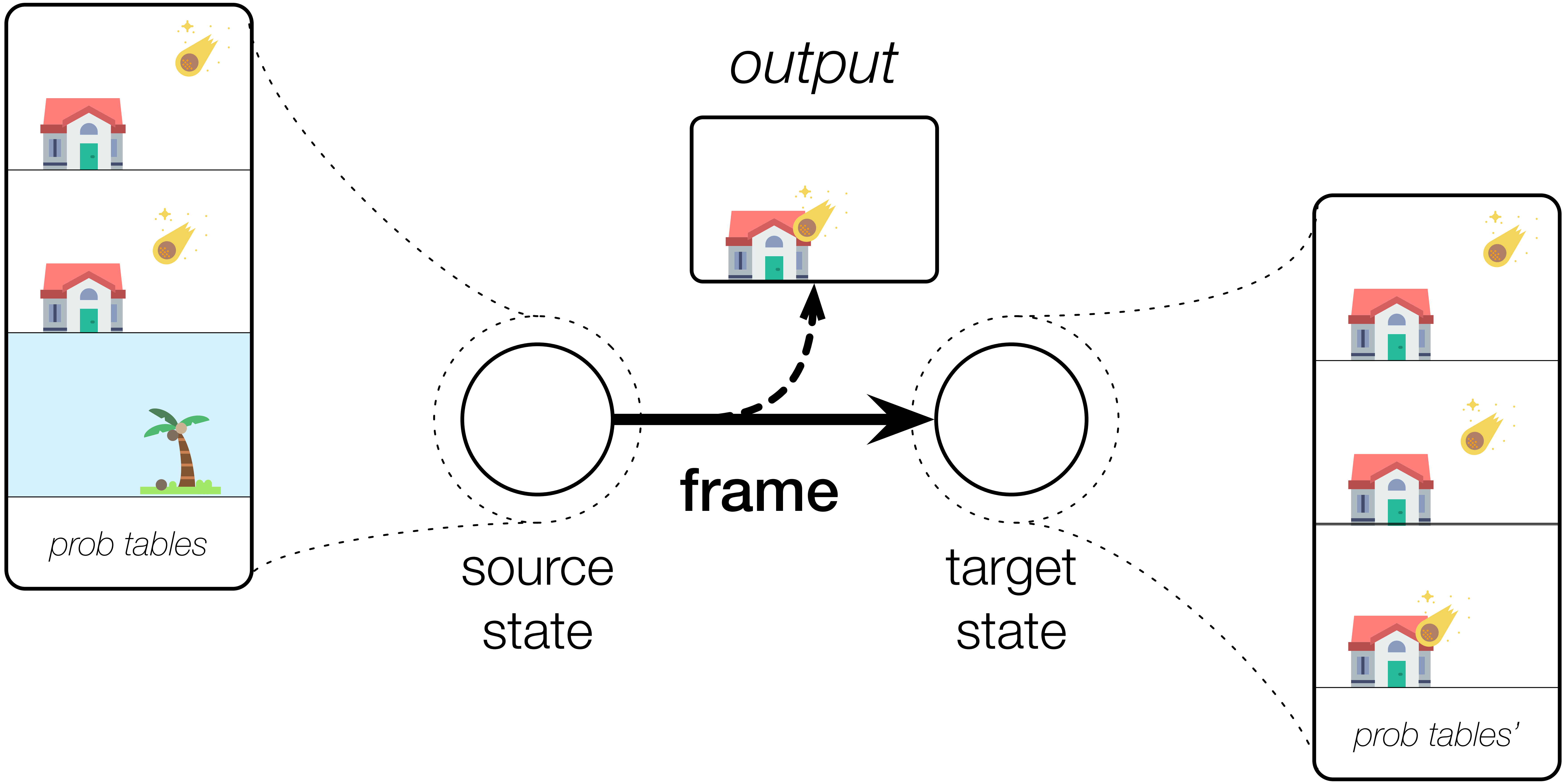
We need a way to start encoding mid-stream

- Start encoding mid-stream needs access to intermediate computations.
- Traditional video codecs *do not* expose this information.
- We formulated this internal information and we made it explicit: the **“state”**.

The decoder is an automaton



The state is consisted of reference images and probability models



What we built: a video codec in explicit state-passing style

- VP8 decoder with no inner state:

```
decode(state, frame) → (state′, image)
```

- VP8 encoder: resume from specified state

```
encode(state, image) → interframe
```

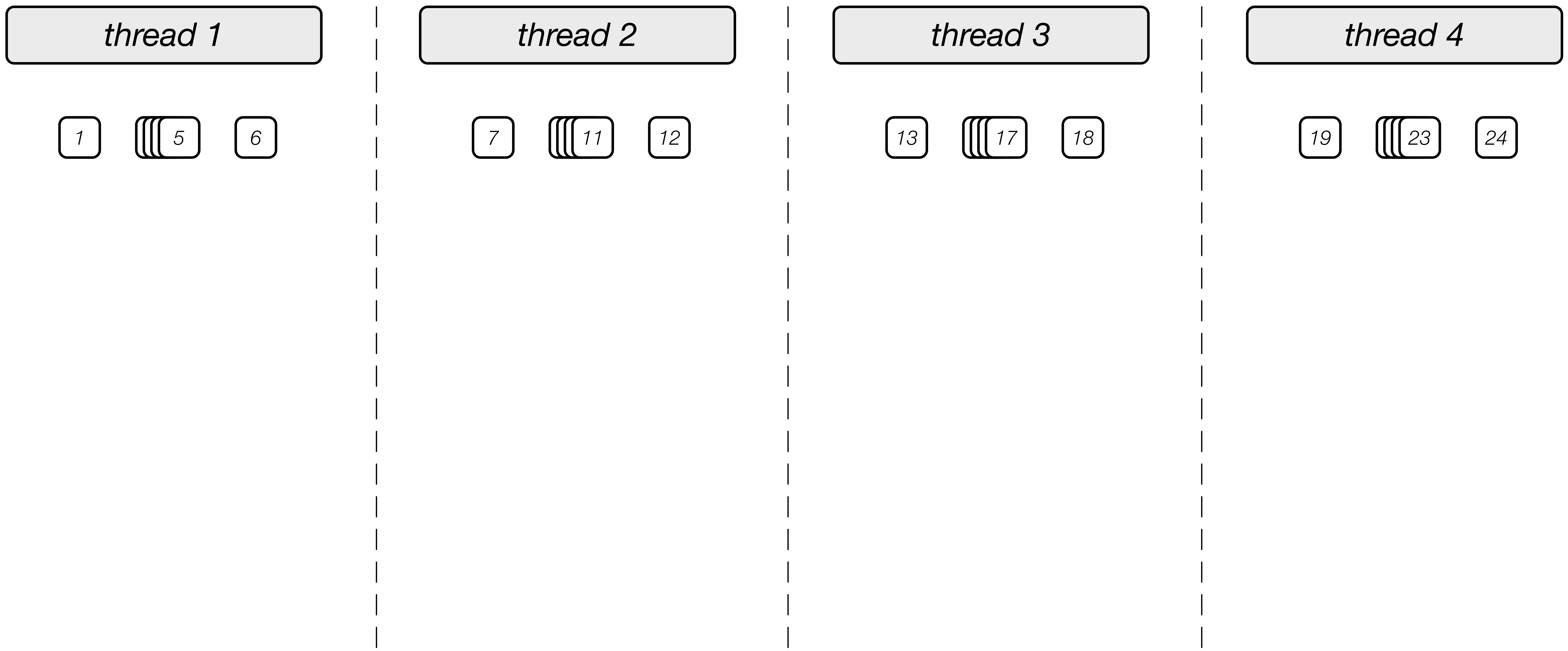
- Adapt a frame to a different source state

```
rebase(state, image, interframe) → interframe′
```

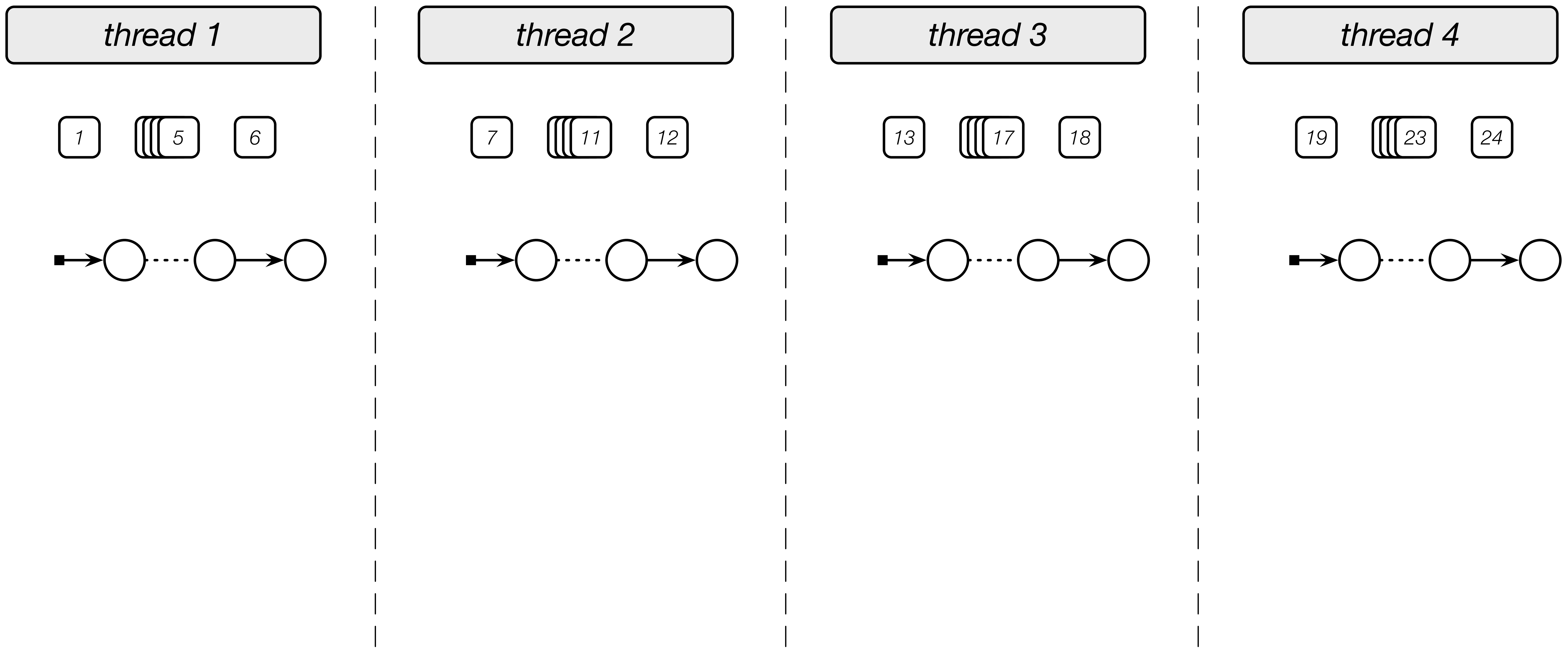
Putting it all together: ExCamera

- Divide the video into tiny chunks:
  - [Parallel] `encode` tiny independent chunks.
  - [Serial] `rebase` the chunks together and remove extra keyframes.

1. [Parallel] Download a tiny chunk of raw video



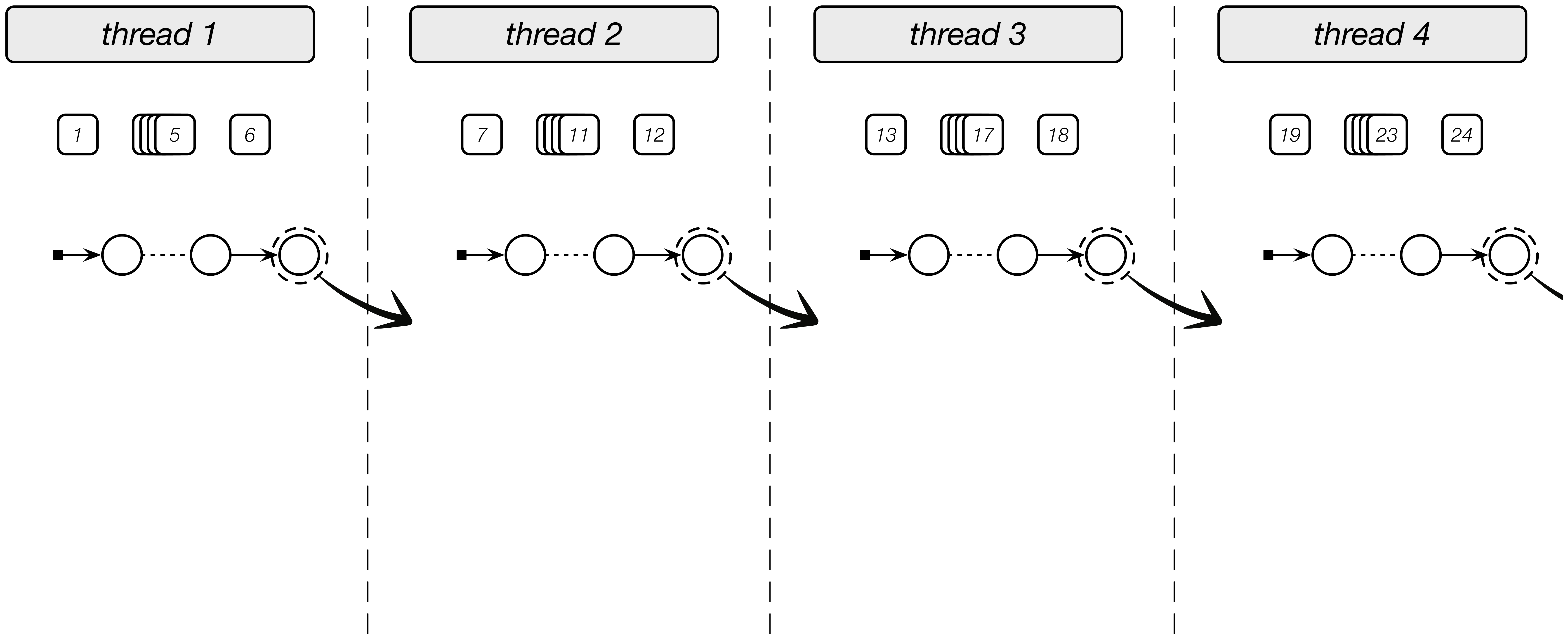
2. [Parallel] `vpxenc` → keyframe, interframe[2:n]



Google's VP8 encoder

`encode(img[1:n]) → keyframe + interframe[2:n]`

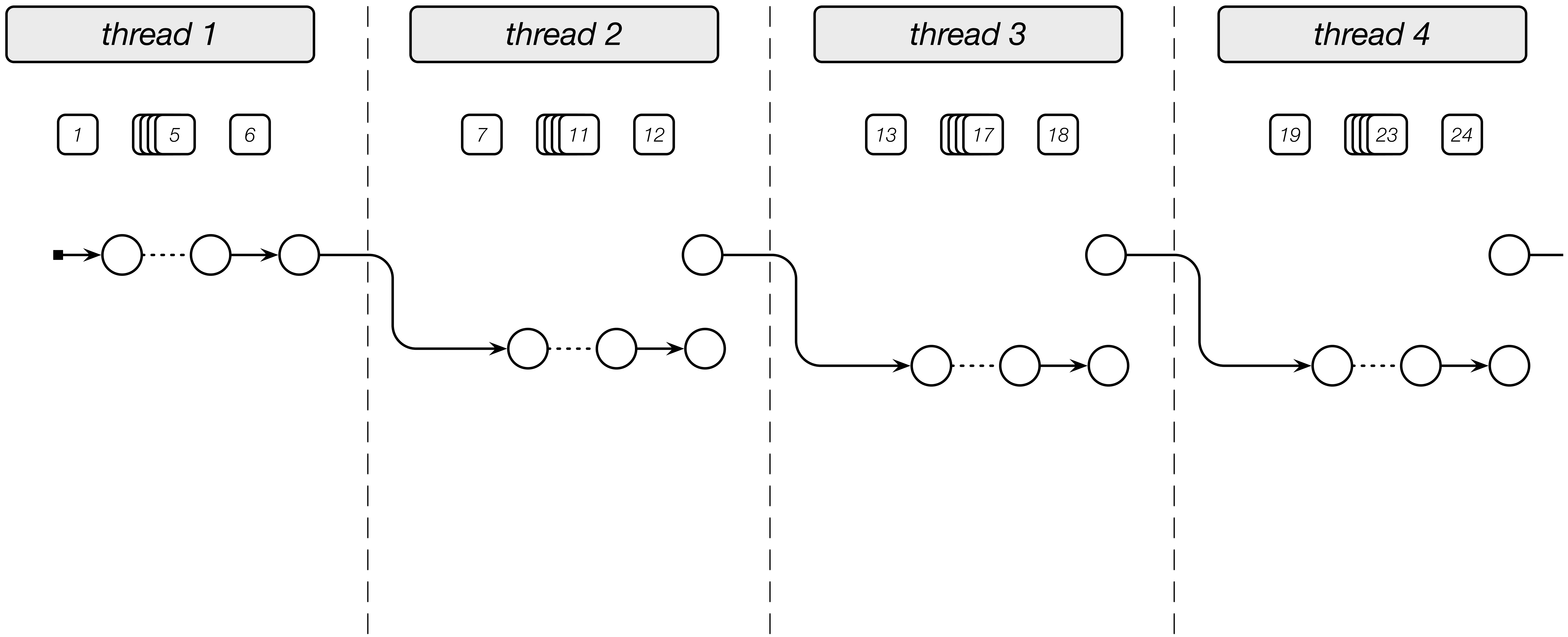
3. [Parallel] decode → state ⇝ *next thread*



Our explicit-state style decoder

```
decode(state, frame) → (state′, image)
```

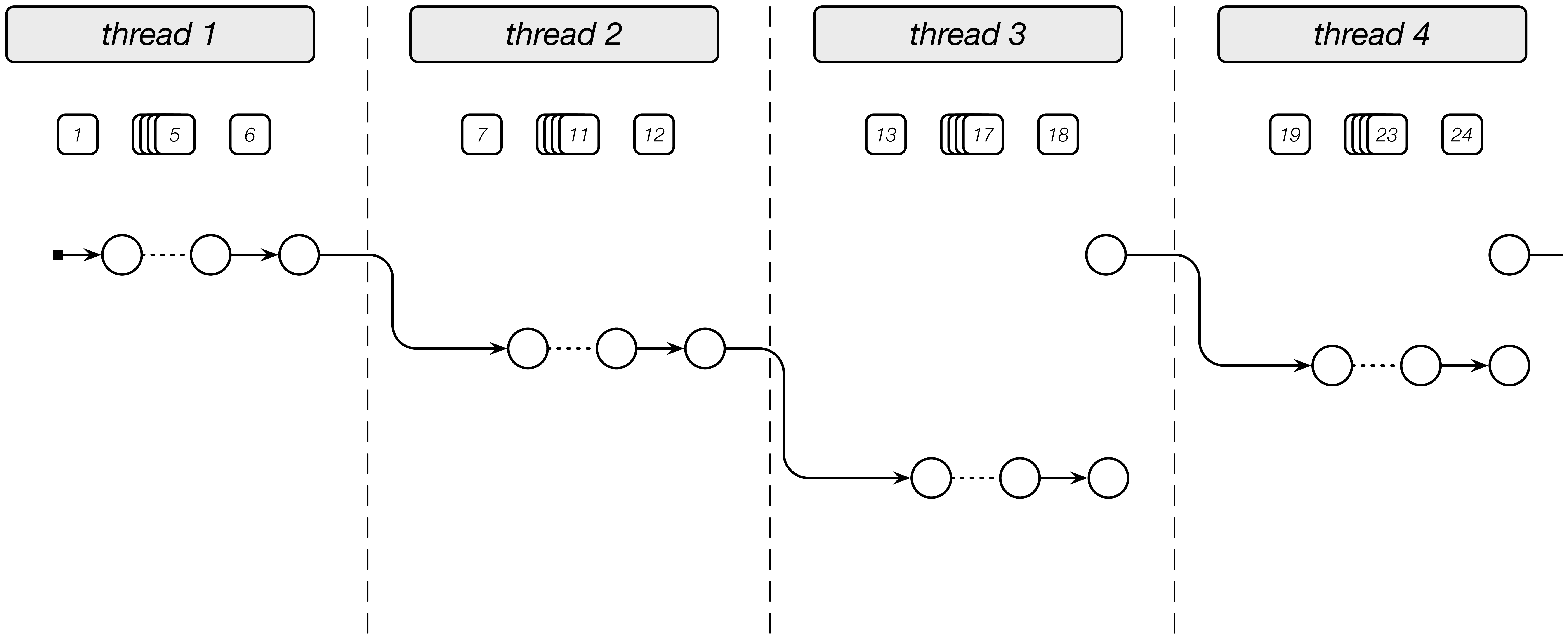
4. [Parallel] *last thread's state* ⇝ encode



Our explicit-state style encoder

```
encode(state, image) → interframe
```

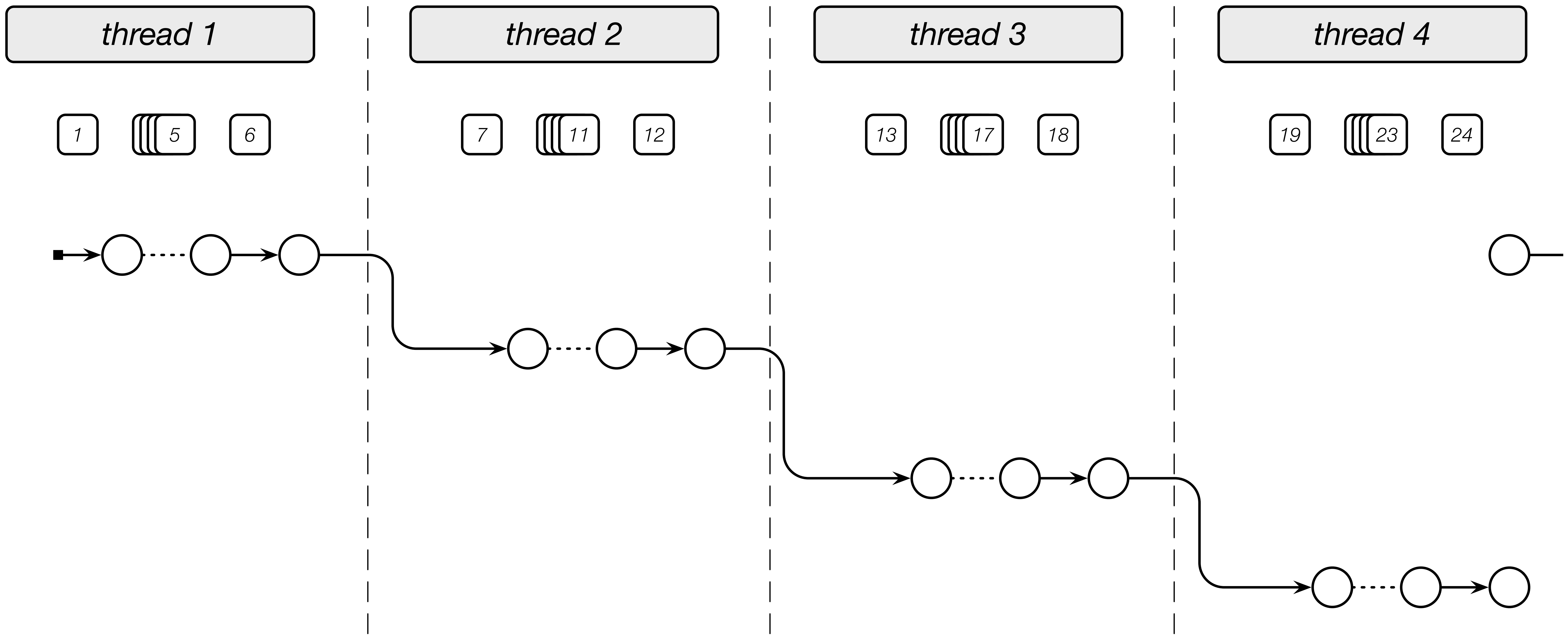
5. [Serial] *last thread's state* → `rebase` → state → *next thread*



Adapt a frame to a different source state

`rebase(state, image, interframe) → interframe′`

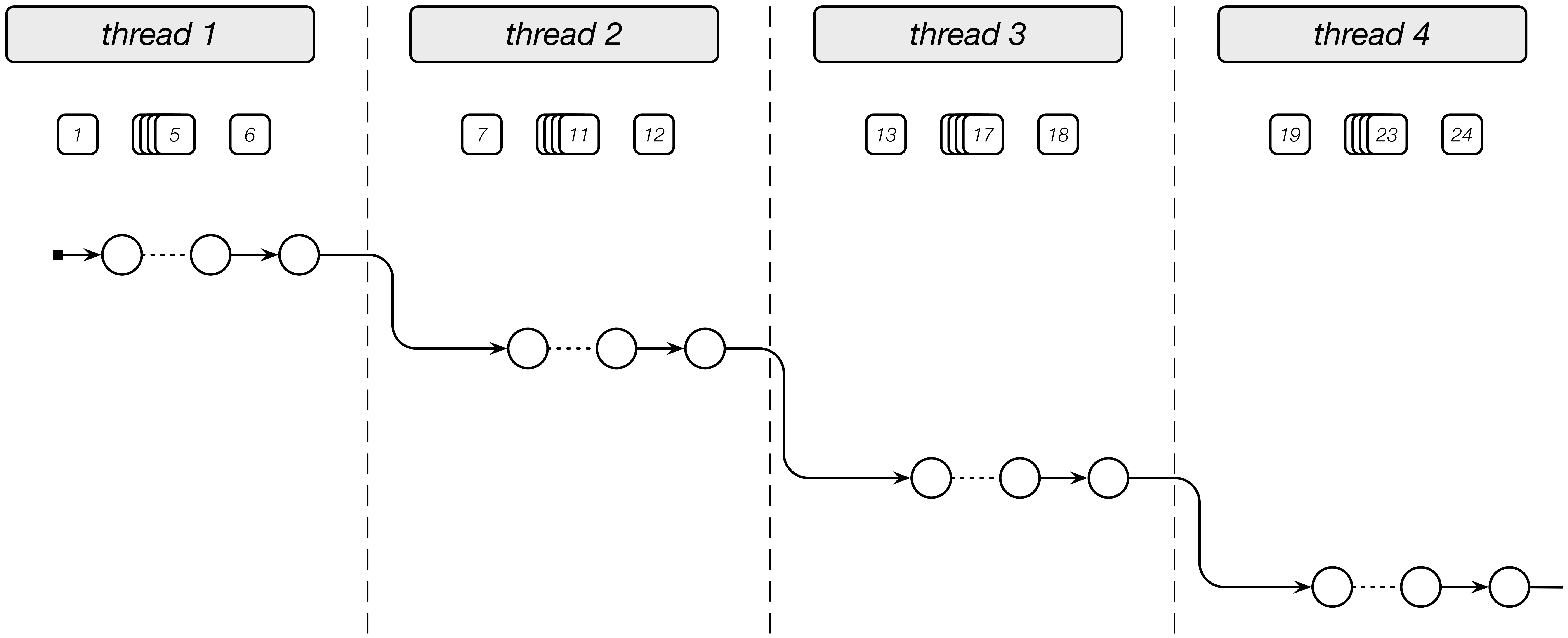
5. [Serial] *last thread's state* → `rebase` → state → *next thread*



Adapt a frame to a different source state

```
rebase(state, image, interframe) → interframe′
```

6. [Parallel] **Upload finished video**



Wide range of different configurations

ExCamera[n, x]

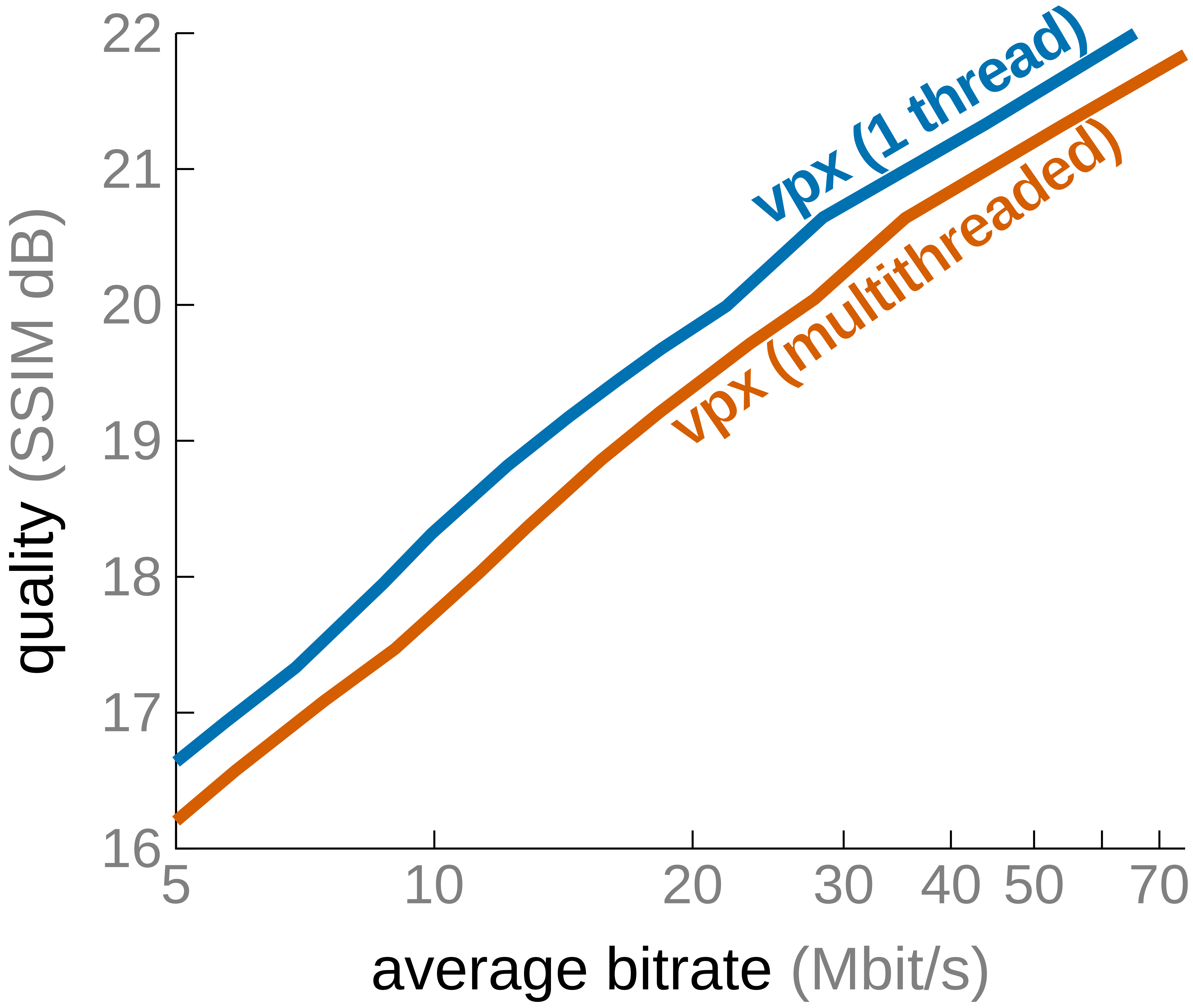
number of frames in each chunk

Wide range of different configurations

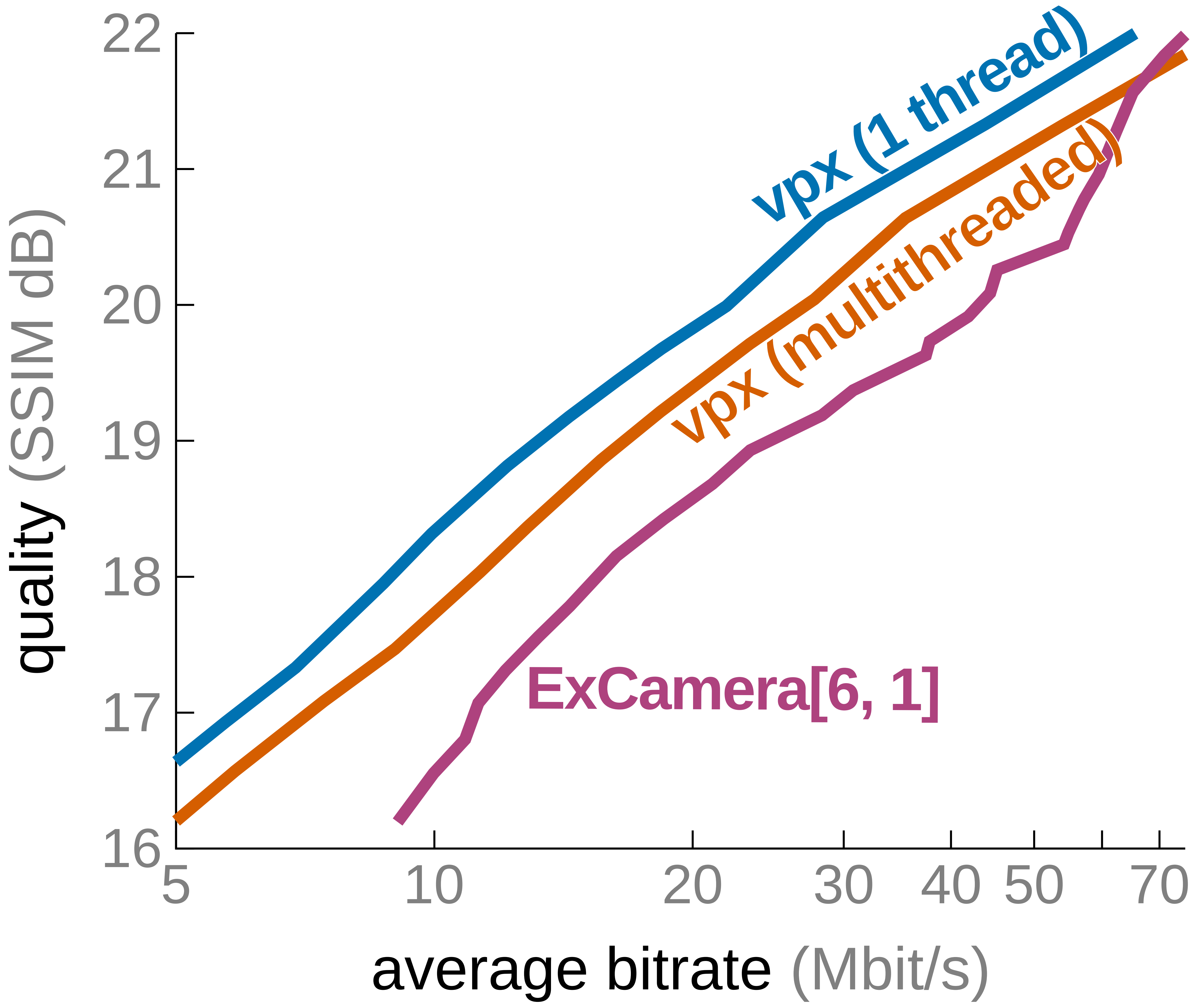
ExCamera[n, x]

number of chunks "rebased" together

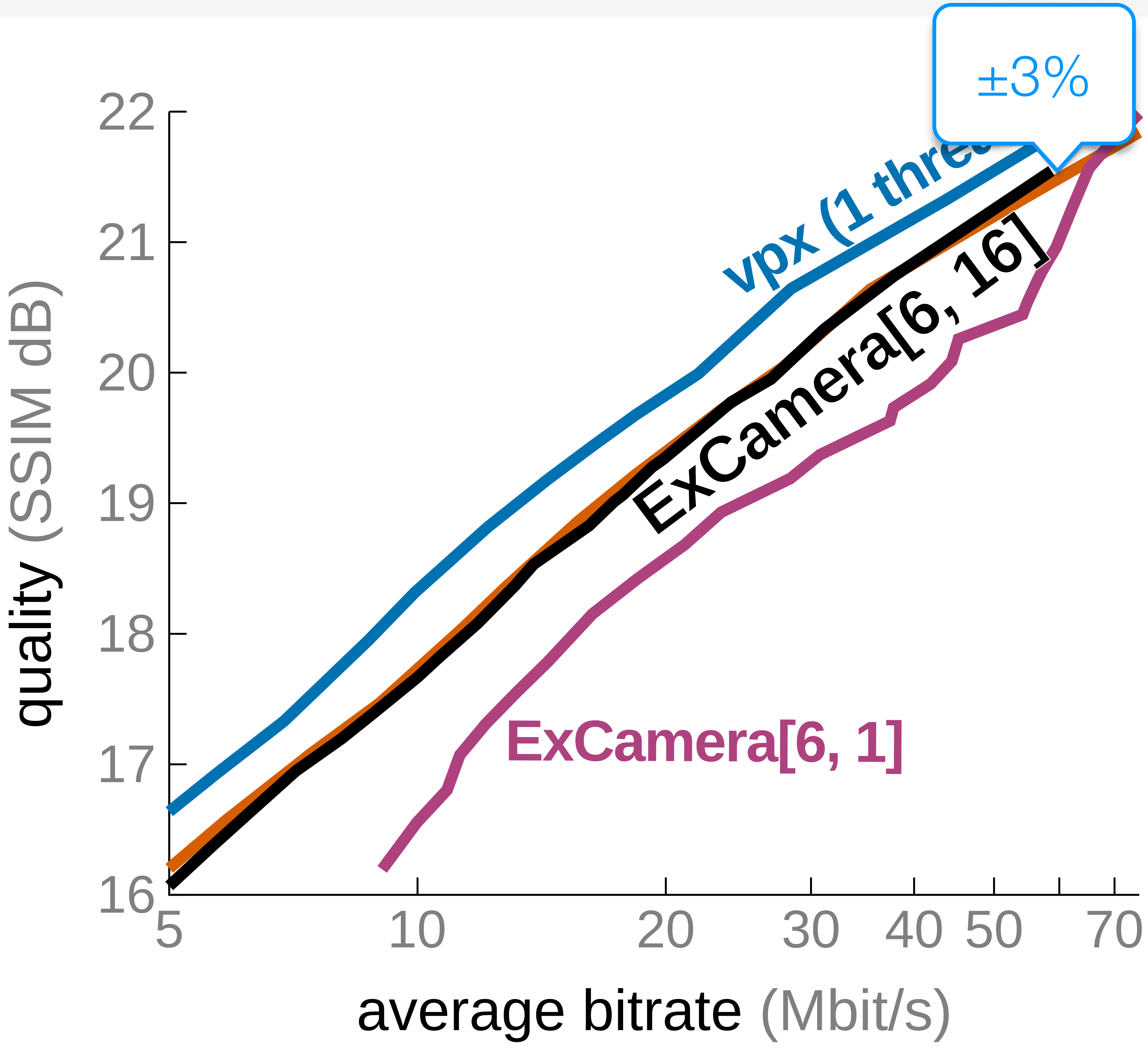
How well does it compress?



How well does it compress?



How well does it compress?



14.8-minute **4K** Video *@20dB*	
vpxenc Single-Threaded	**453 mins**
vpxenc Multi-Threaded	**149 mins**
YouTube (H.264)	**37 mins**
ExCamera[6, 16]	**2.6 mins**

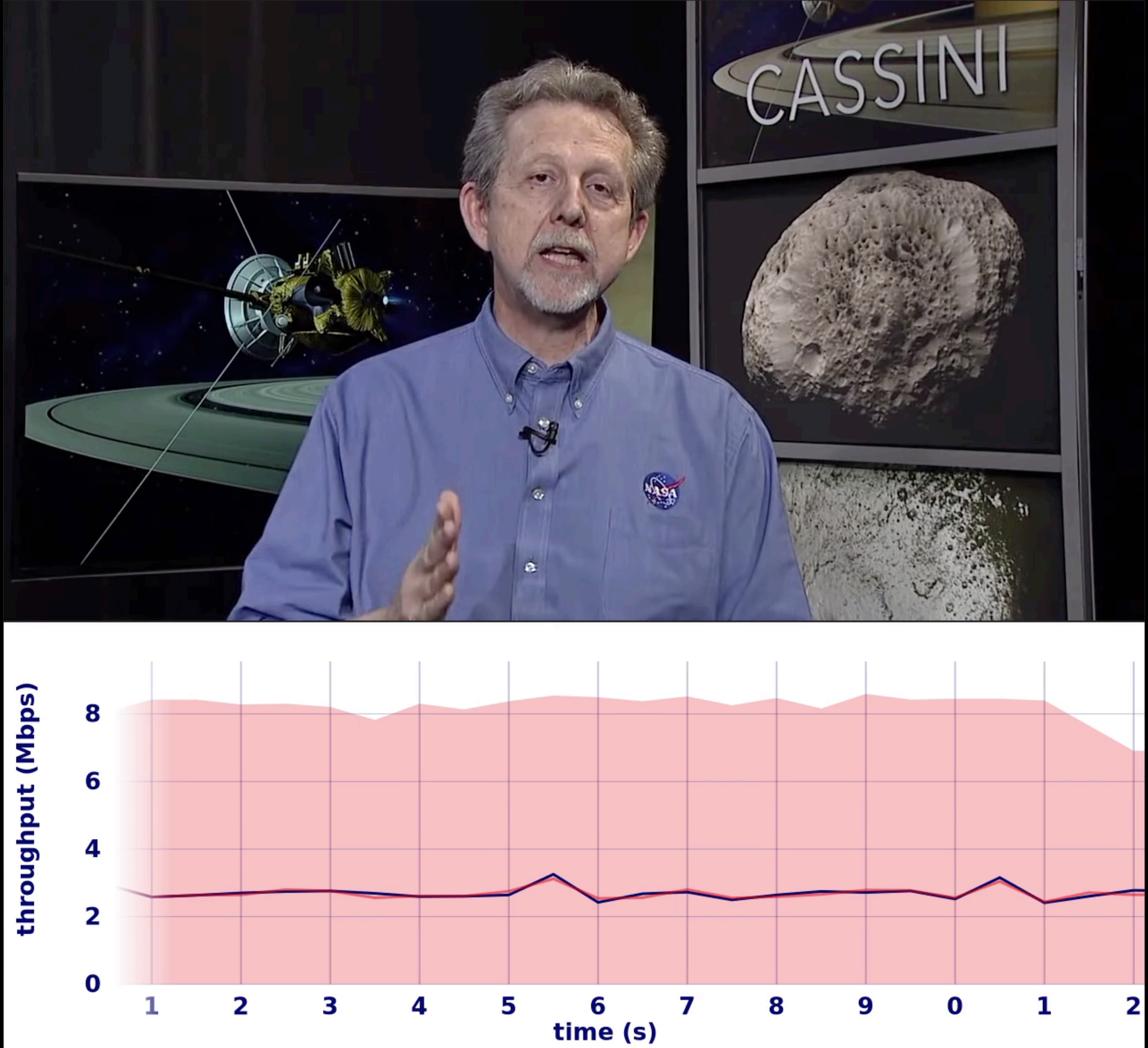
ExCamera concluding thoughts

- Functional video codec lets ExCamera **parallelize** at fine granularity.
- Many interactive jobs call for similar approach:
  - Image and video filters
  - 3D artists
  - Compilation and software testing
  - Interactive machine learning
  - Database queries
  - Data visualization
  - Genomics
  - Search
- Distributed systems will need to treat application state as a first-class object.
- Every program soon: **do in 1 hour** **do in 1 second for 9¢**

System 3: Salsify (videoconferencing)

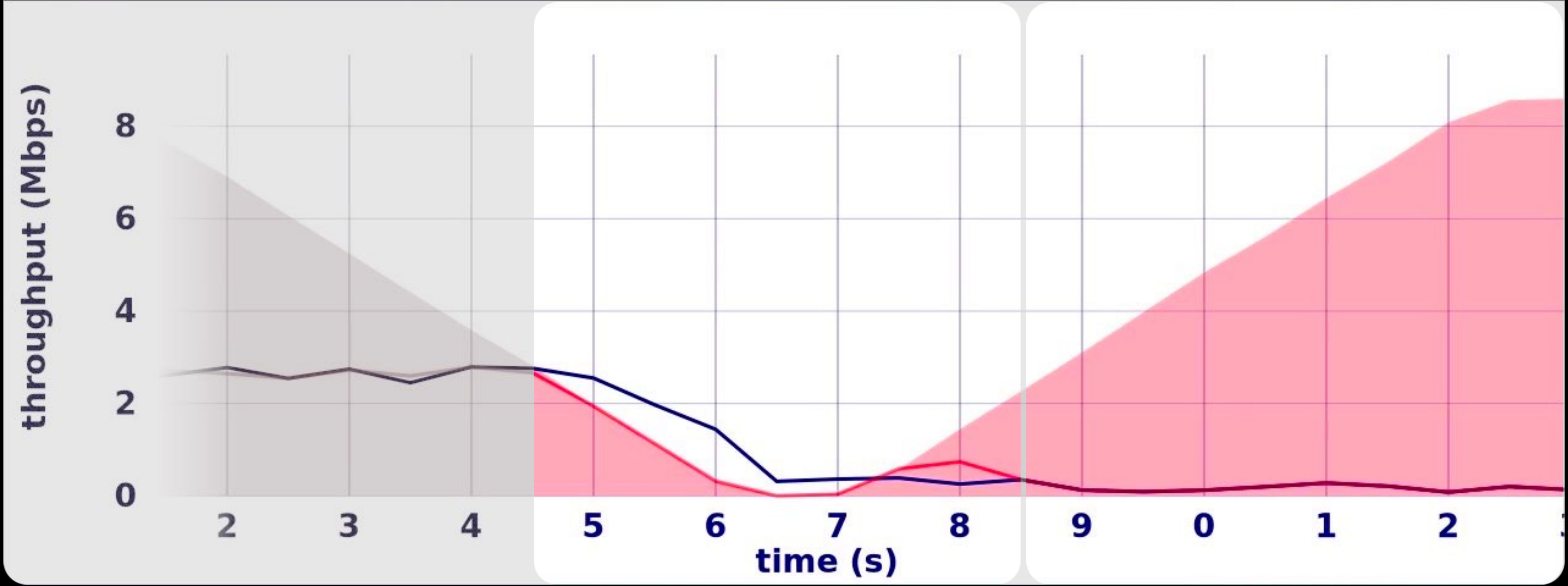
Sadjad Fouladi, John Emmons, Emre Orbay, Catherine Wu, Riad S. Wahby, and KW, **Salsify: low-latency network video through tighter integration between a video codec and a transport protocol**, in NSDI 2018.

https://snr.stanford.edu/salsify

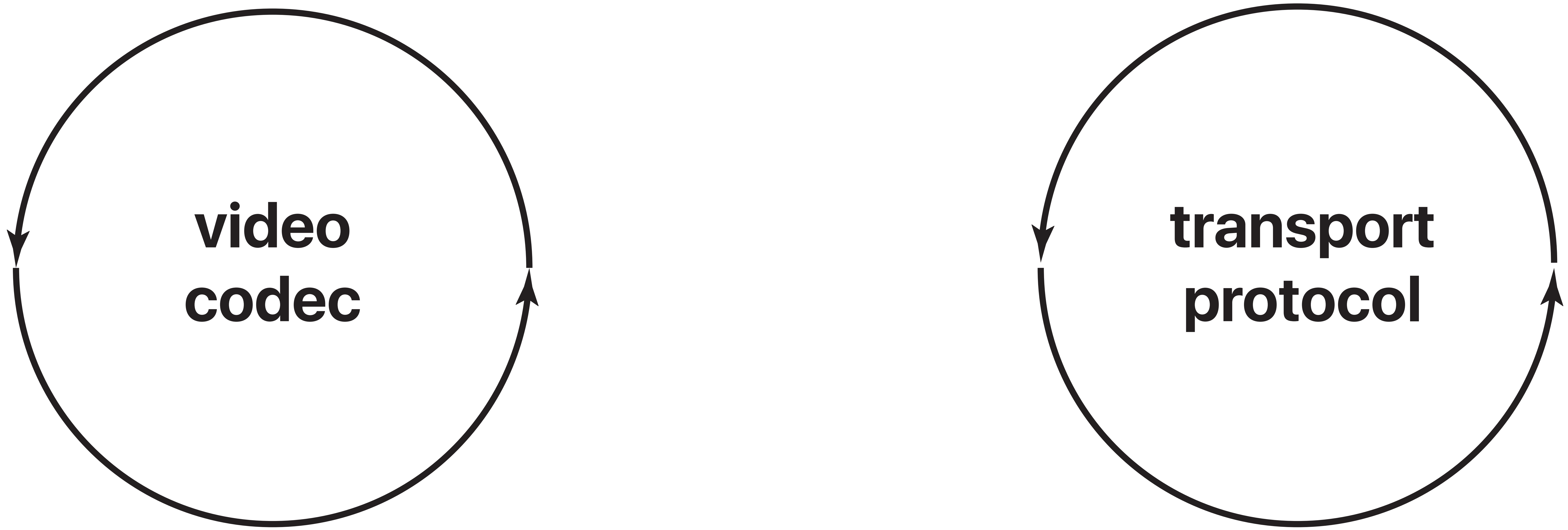


WebRTC
(Chrome 65)

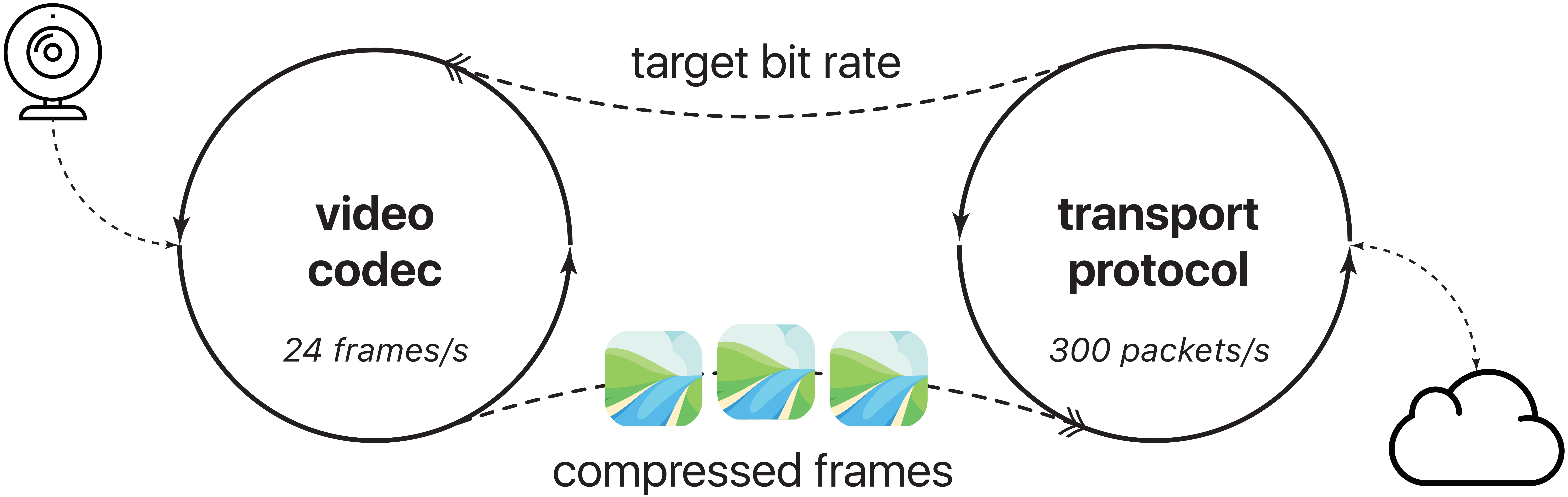
Current systems do not react **fast enough** to **network variations**, end up congesting the network, causing **stalls and glitches**.



Today's systems combine two *(loosely-coupled)* components



Two distinct modules, two separate control loops



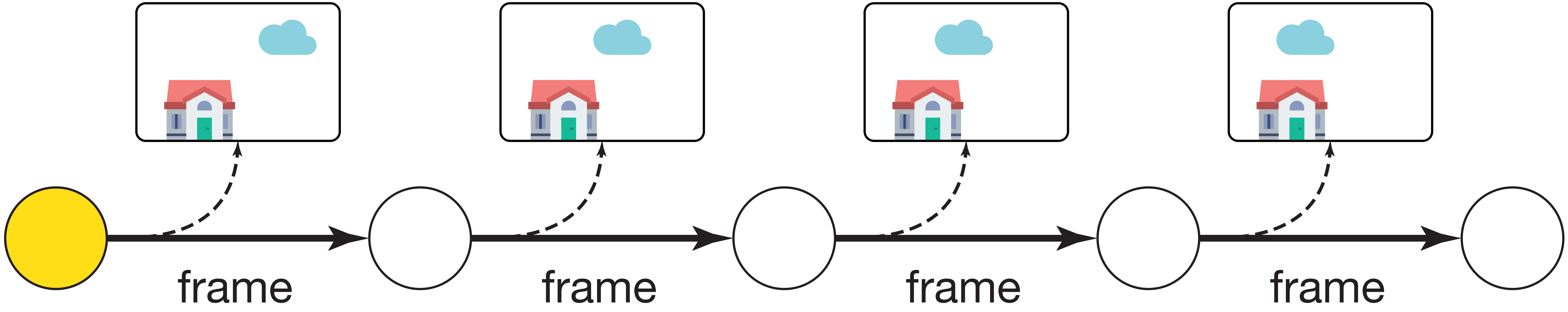
Transport tells us how big the next frame should be, but...

It's challenging for **any codec** to choose the appropriate quality settings upfront to meet a **target size**—they tend to over-/undershoot the target.

How to get an accurate frame out of an inaccurate codec

- **Trial and error:** Encode with different quality settings, pick the one that fits.
  - *Not possible with existing codecs.*

After encoding a frame, the encoder goes through a state transition that is impossible to undo



There's no way to undo an encoded frame in current codecs

`encode(🏞️,🏞️,...) → frames...`

The state is internal to the encoder—no way to save/restore the state.

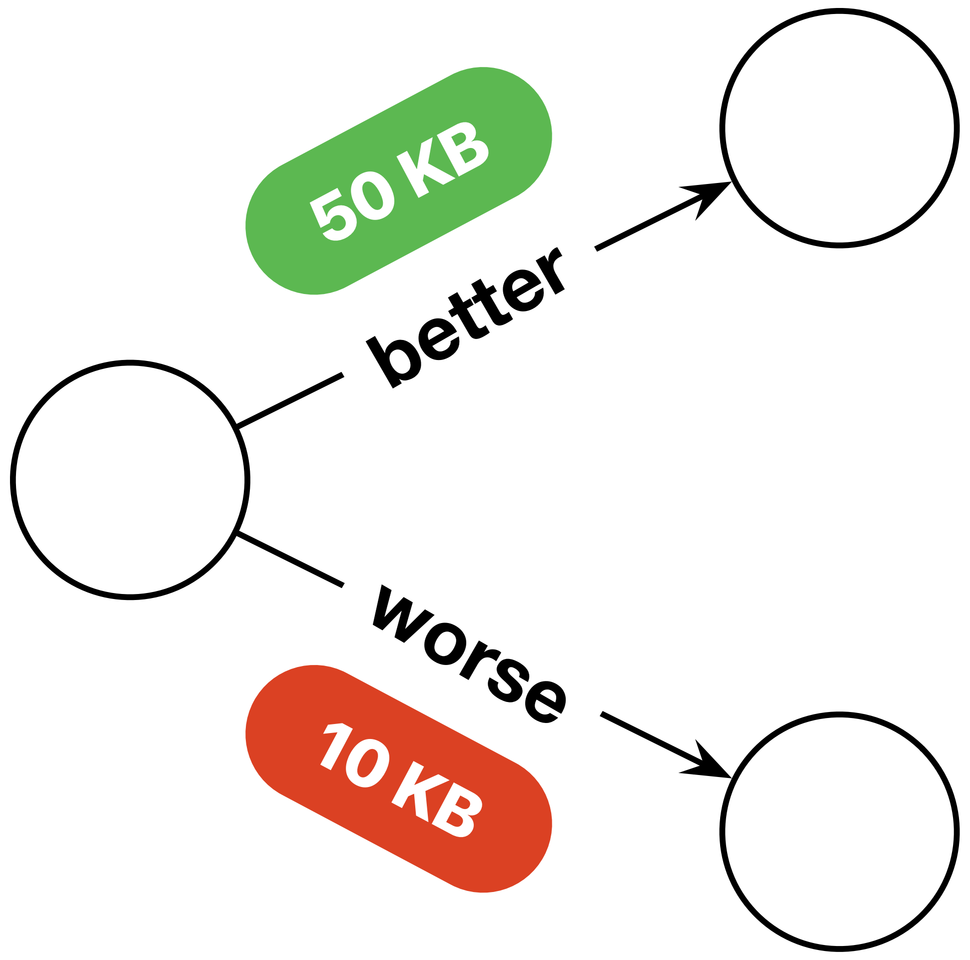
Functional video codec to the rescue

encode(*state*, 🏞) → *state′*, frame

Salsify's functional video codec exposes the state that can be saved/restored.

Order two, pick the one that fits!

- Salsify's functional video codec can **explore different execution paths** without committing to them.
- For each frame, codec presents the transport with *three* options:
  - 🔼 A slightly-higher-quality version,
  - 🔽 A slightly-lower-quality version,
  - ✖ Discarding the frame.

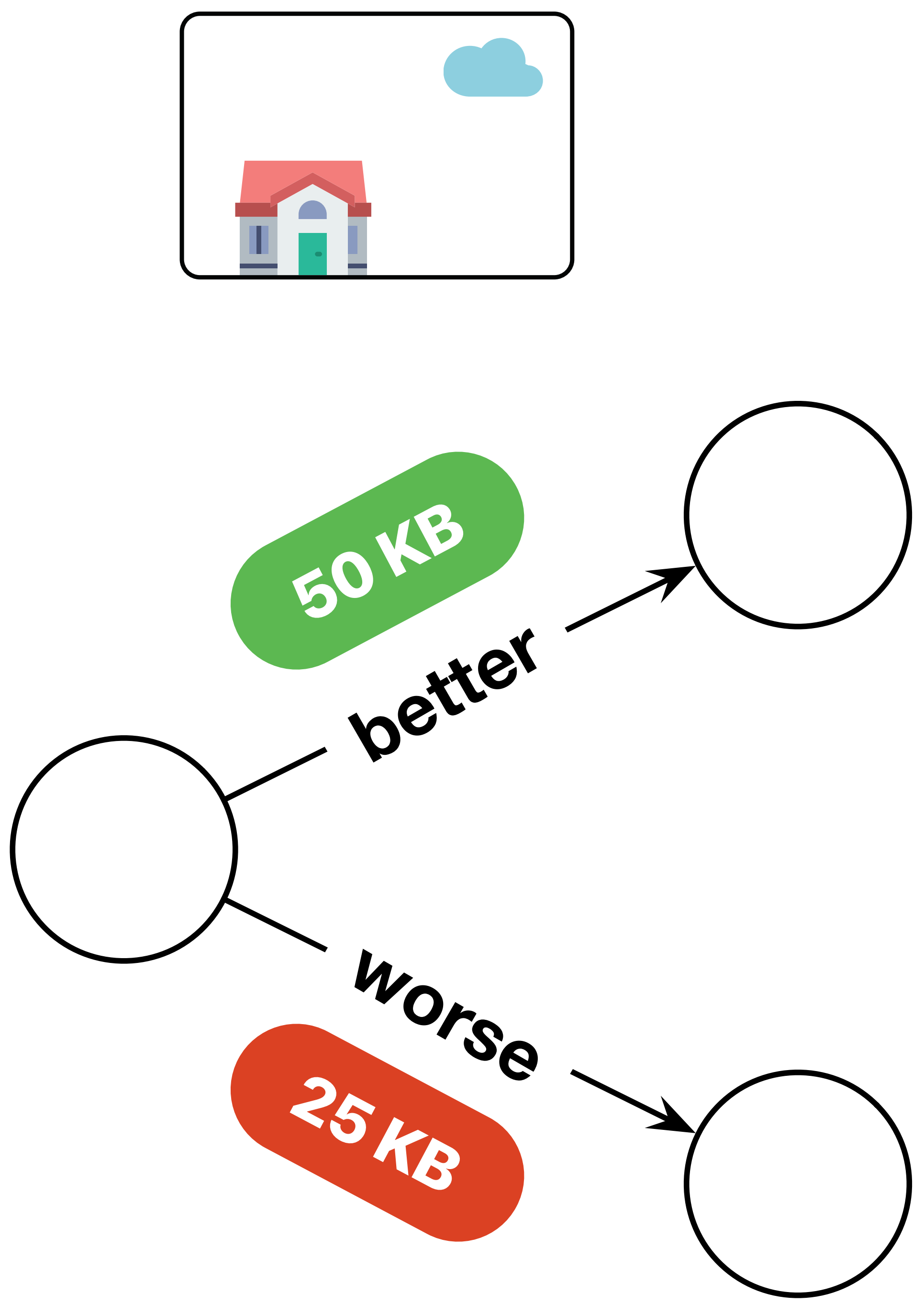


Salsify's architecture:

Unified control loop

Codec → Transport

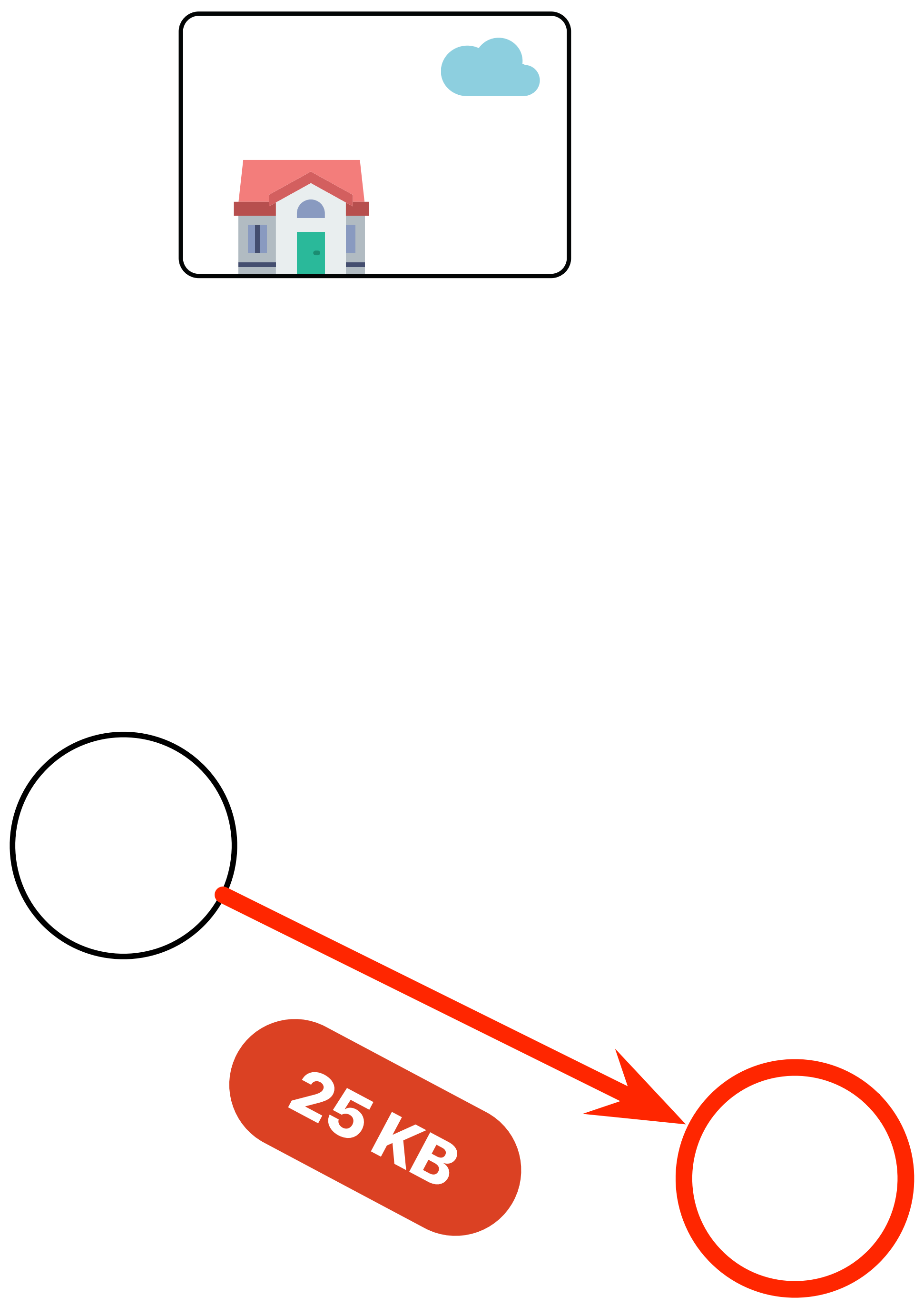
“Here’s two versions of the current frame.”



target frame size **30 KB**

Transport → Codec

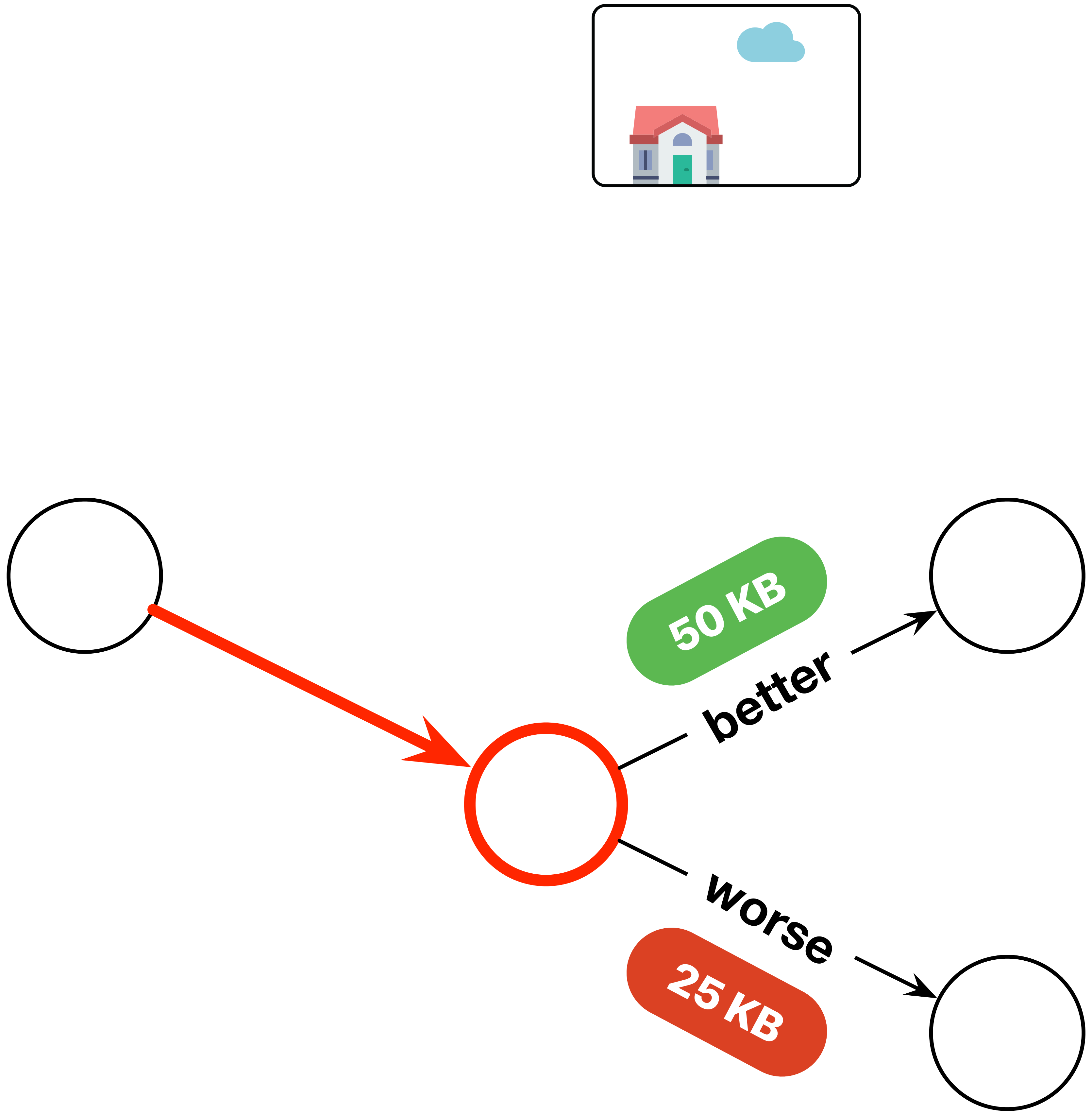
“I picked option 2. Base the next frame on its exiting state.”



target frame size 30 KB

Codec → Transport

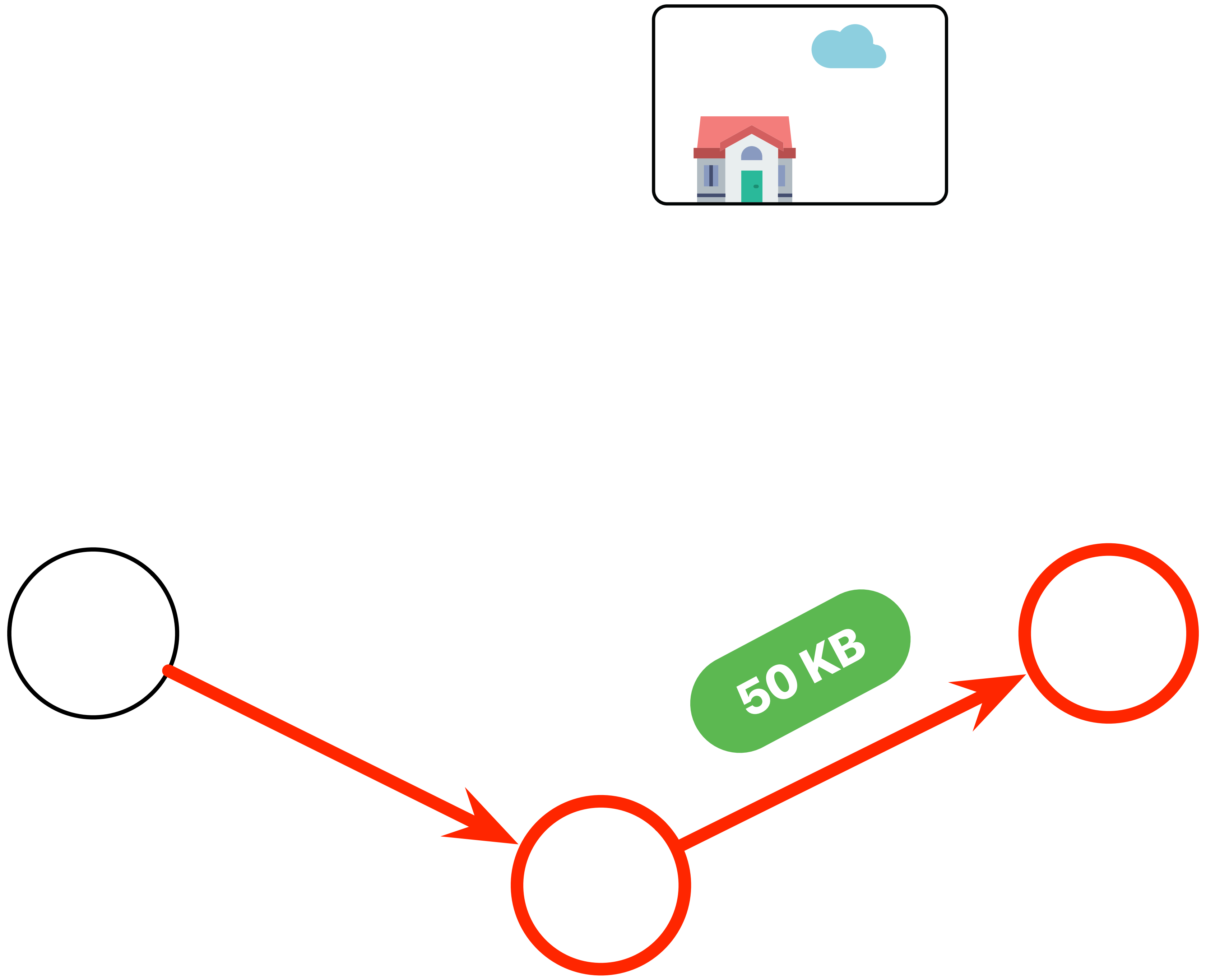
“Here’s two versions of the latest frame.”



target frame size **55 KB**

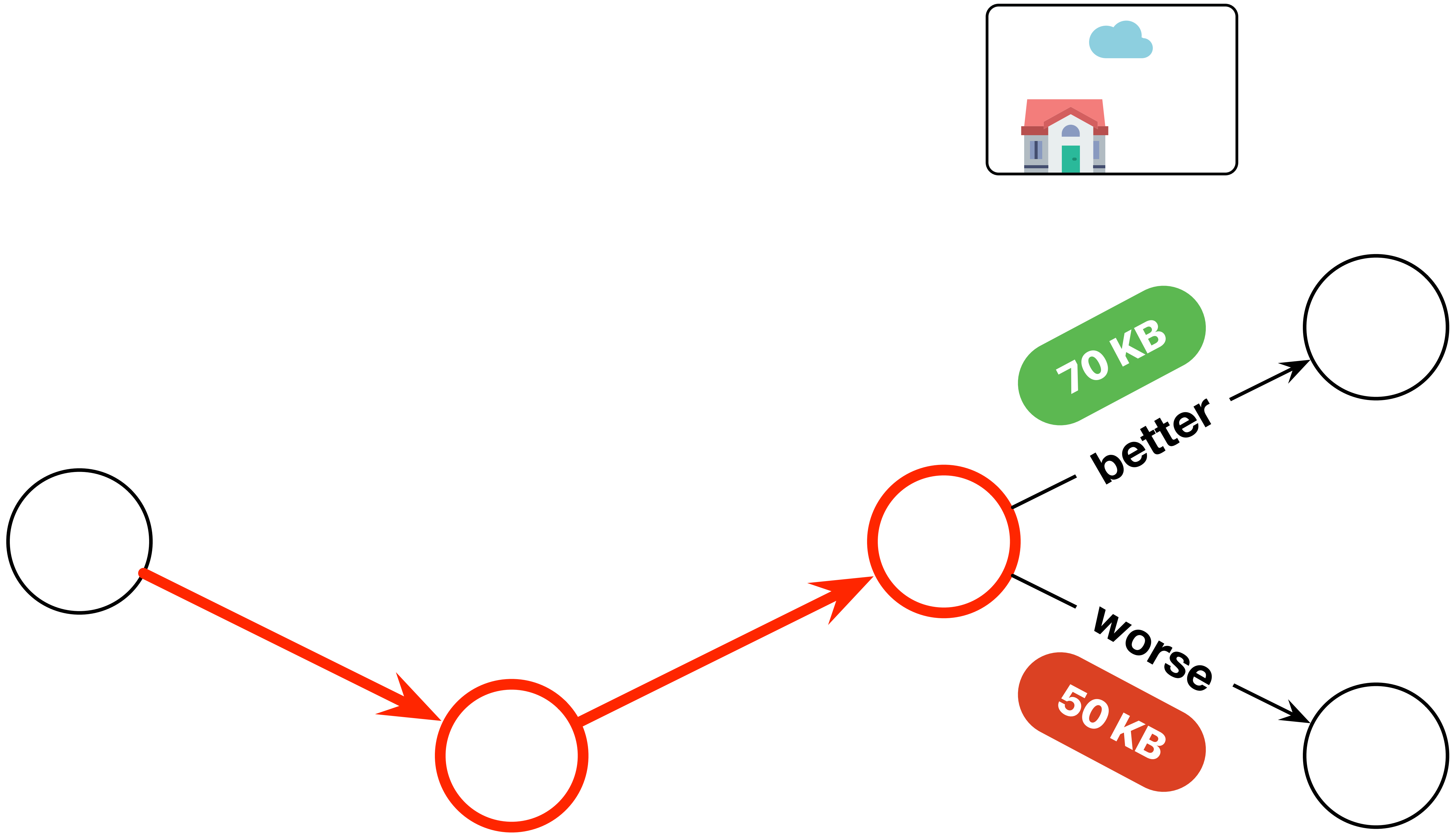
Transport → Codec

“I picked option 1. Base the next frame on its exiting state.”



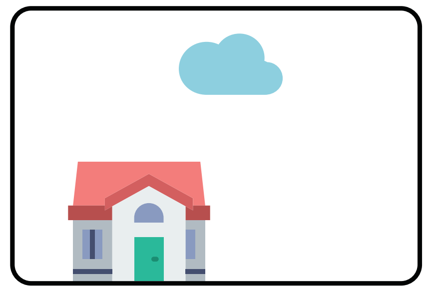
target frame size **55 KB**

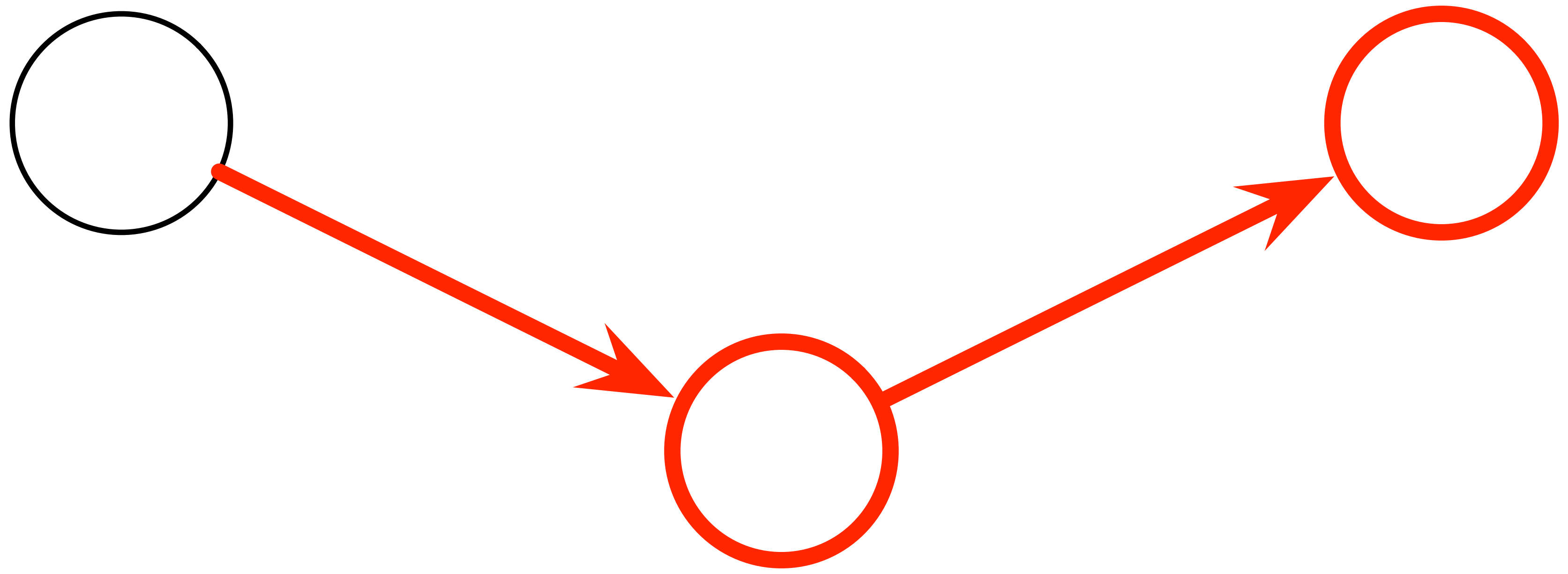
Codec → Transport
“Here’s two versions of the latest frame.”



target frame size **5 KB**

Transport → Codec
“I cannot send any frames right now. Sorry, but discard them.”

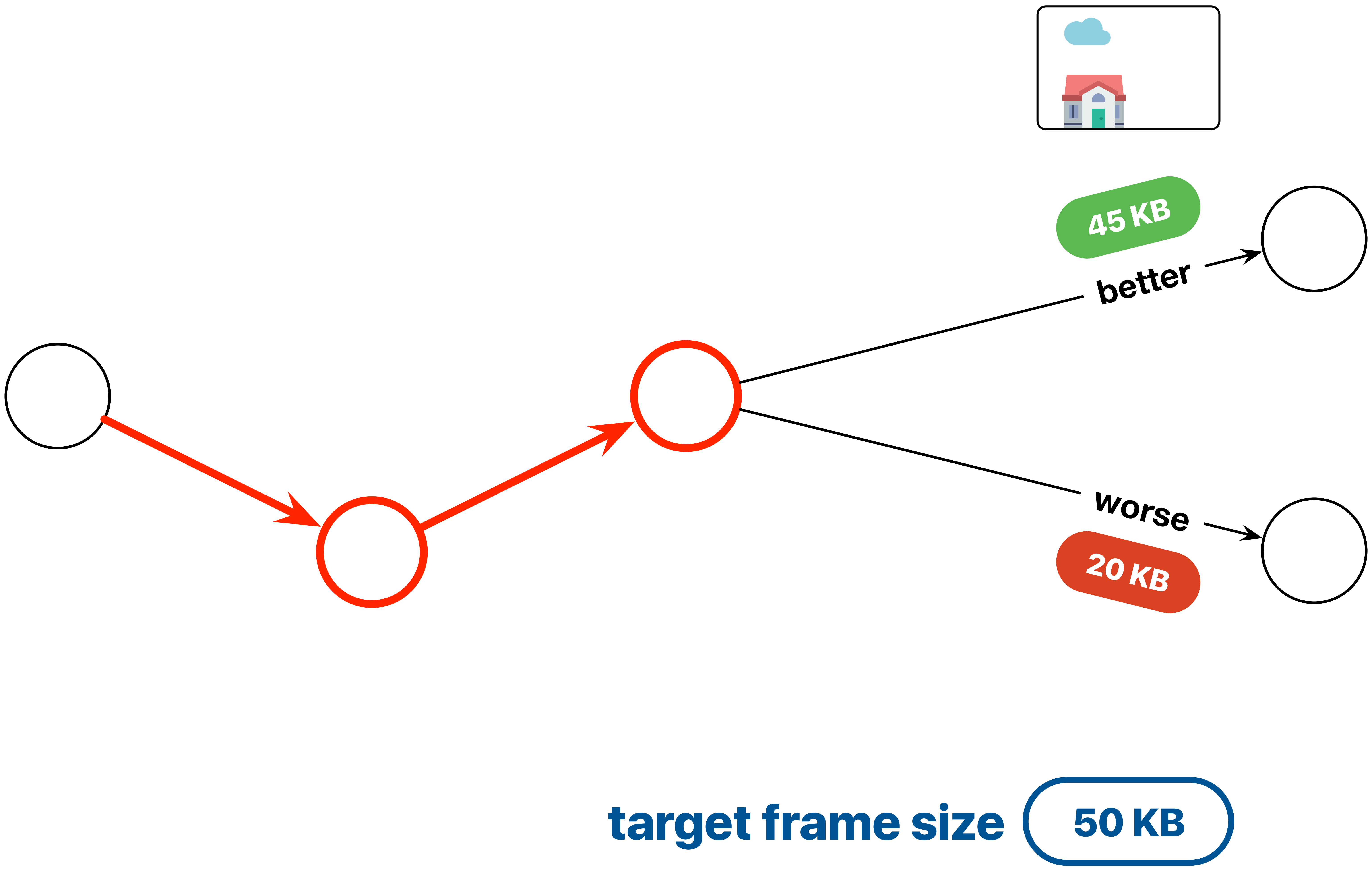




target frame size 5 KB

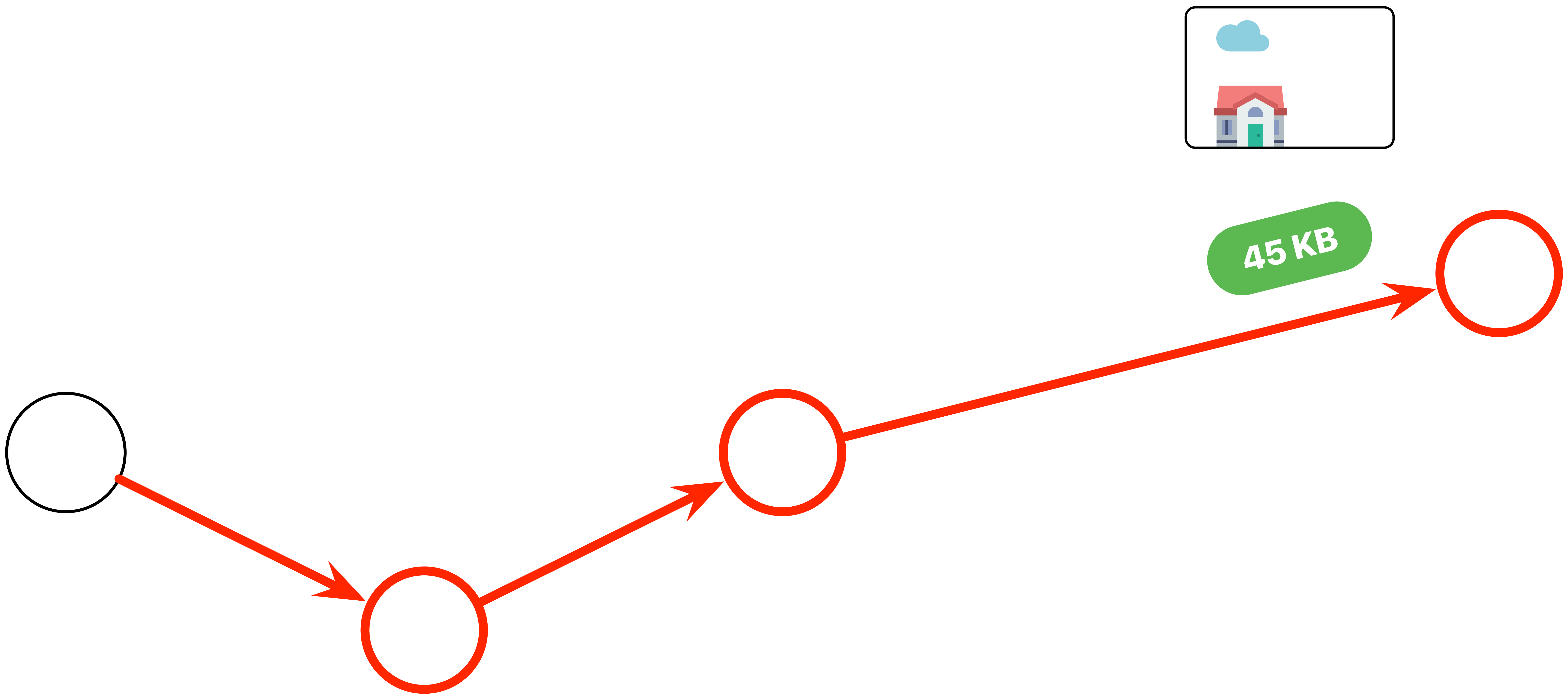
Codec → Transport

“Fine. Here’s two versions of the latest frame.”



Transport → Codec

"I picked option 1. Base the next frame on its exiting state."



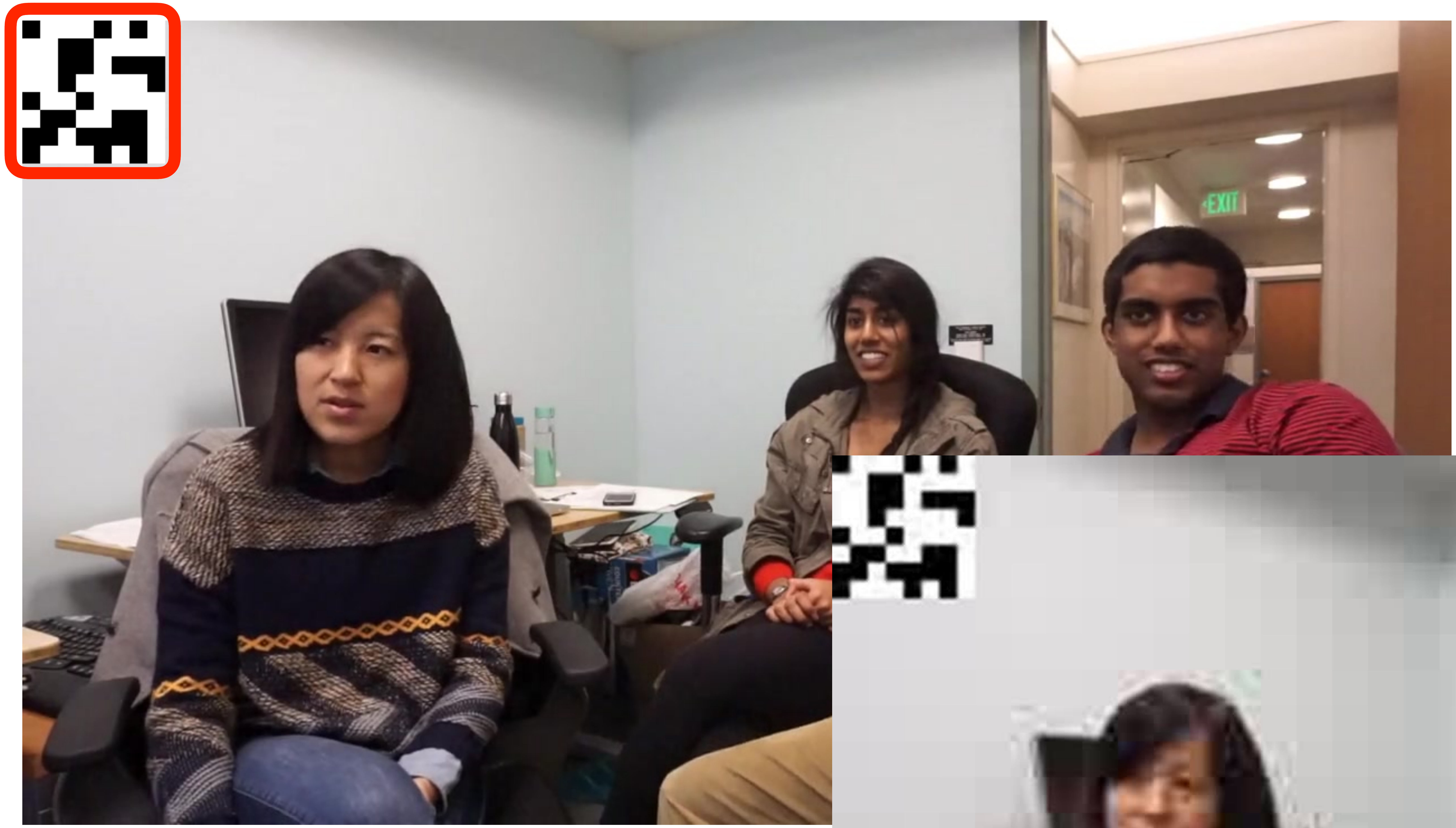
target frame size 50 KB

Goals for the measurement testbed

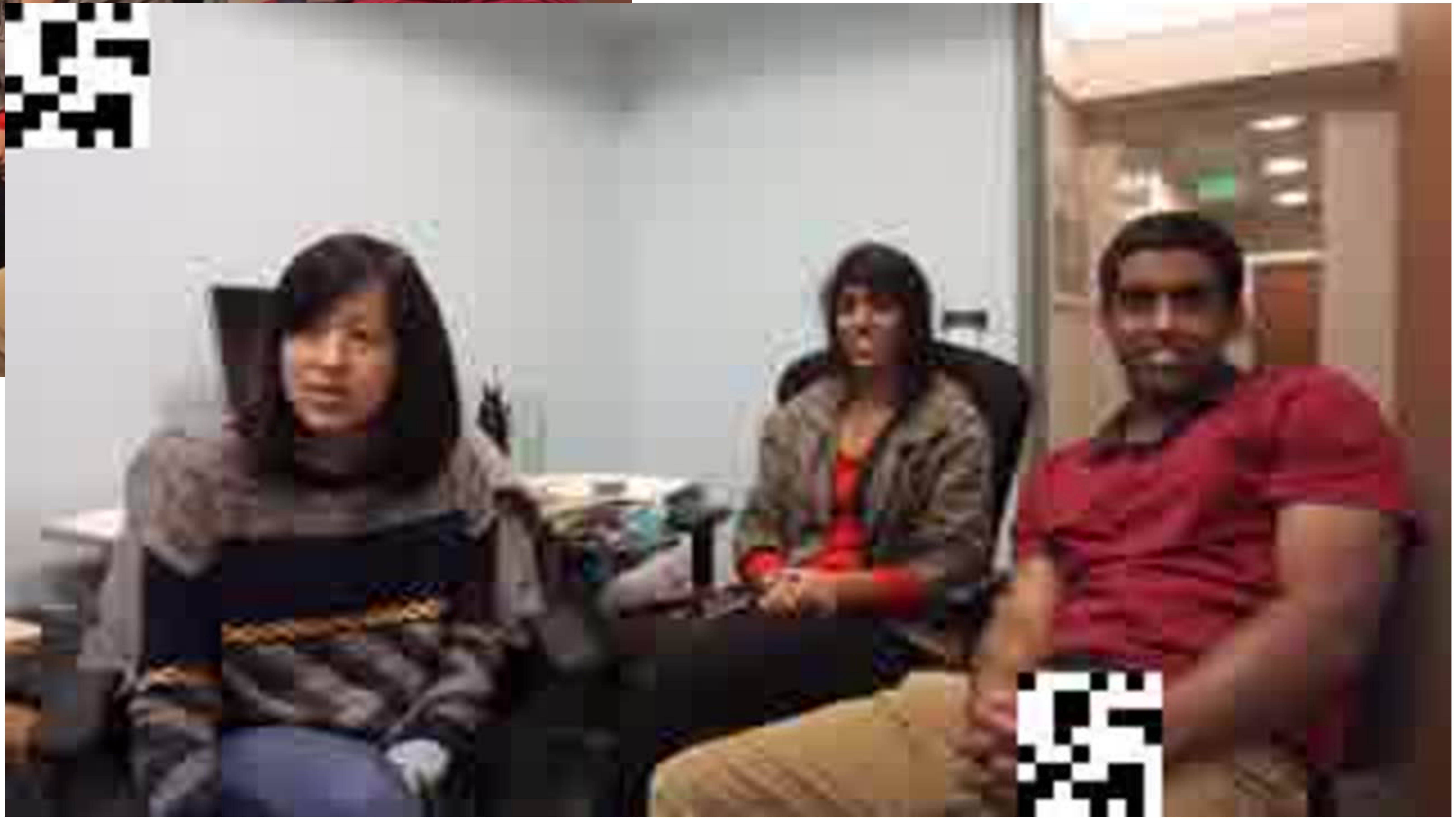
- A system with
  reproducible input video and
  reproducible network traces that runs
  unmodified version of the system-under-test.

- Target QoE metrics: per-frame **quality** and **delay**.


emulated network
barcoded video
receiver HDMI output
video in/out (HDMI)
HDMI to USB camera
Sender
Receiver
Network Simulator
AV.io
Video System
Measurement System

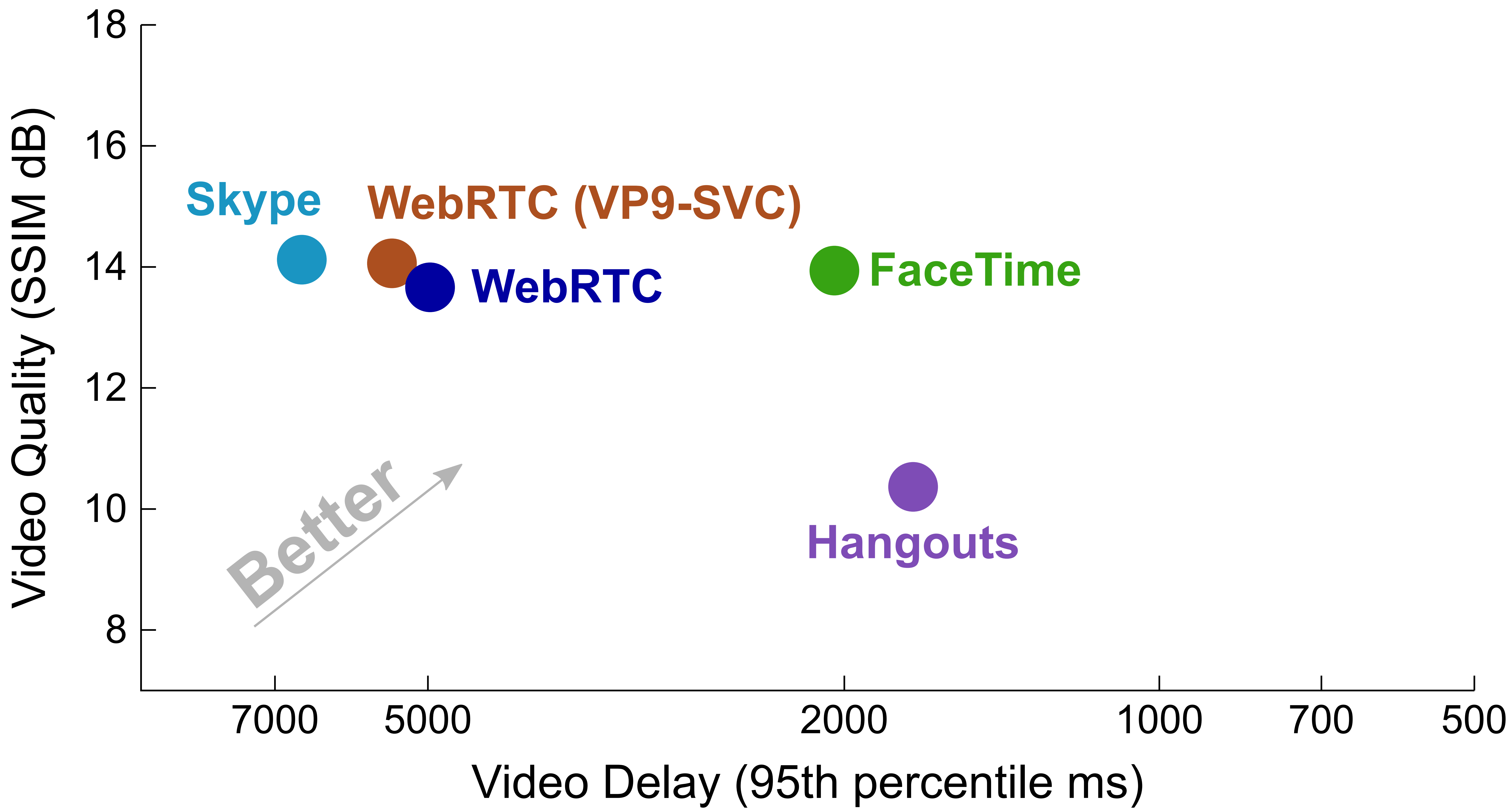


Sent Image
Timestamp: T+0.000s

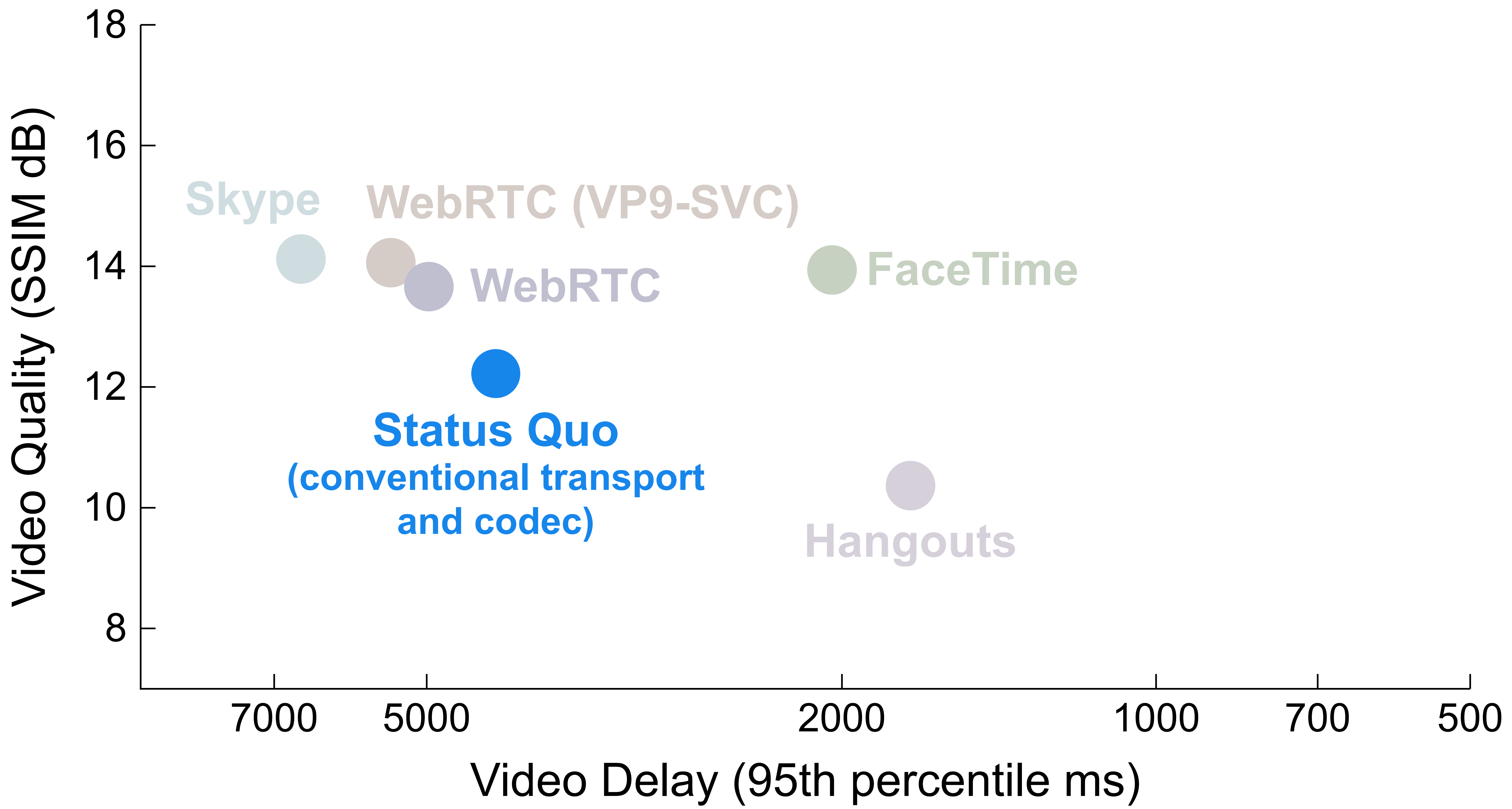


Received Image
Timestamp: T+0.765s
Quality: 9.76 dB SSIM

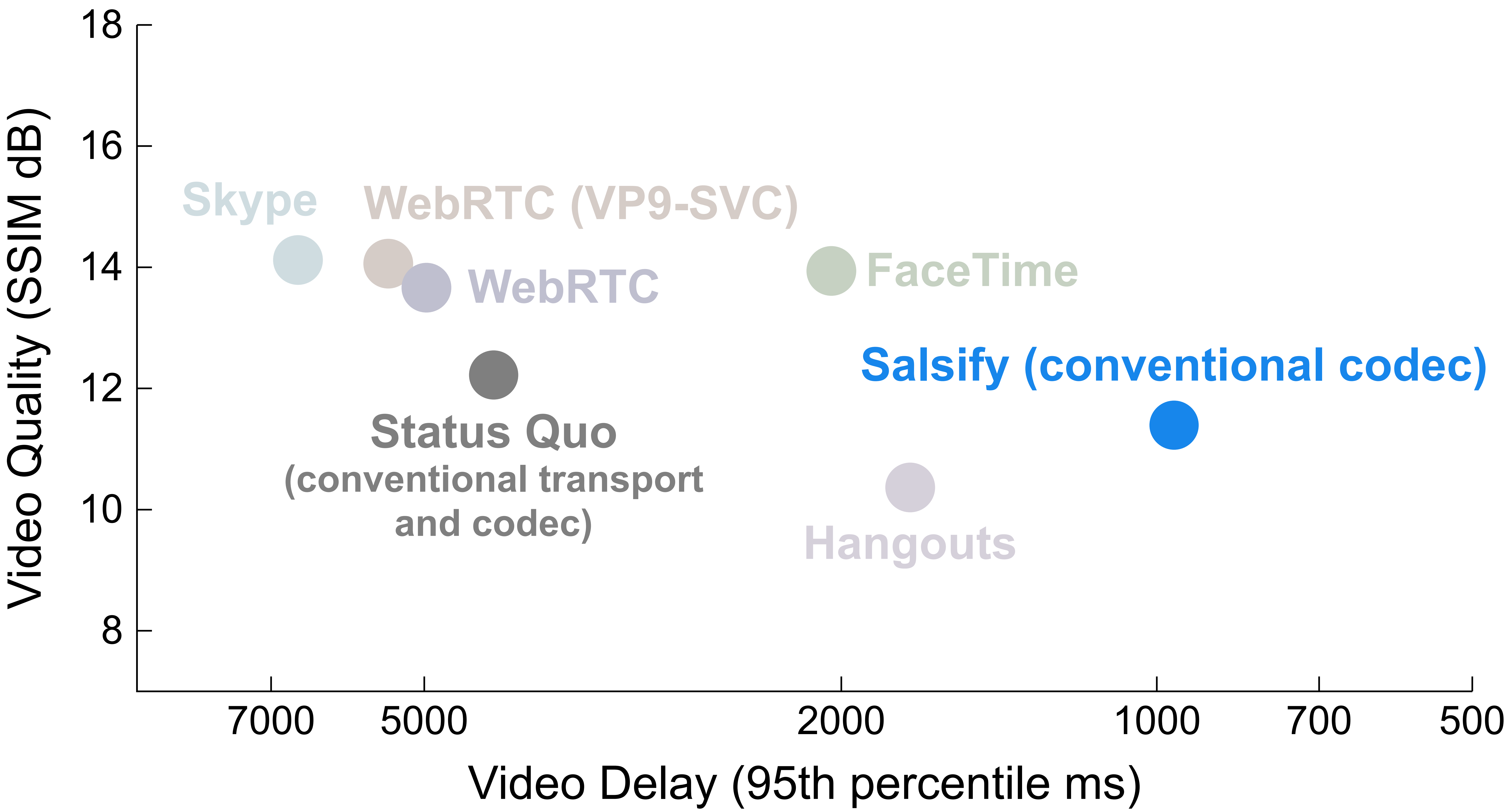
Evaluation results: **Verizon LTE Trace**



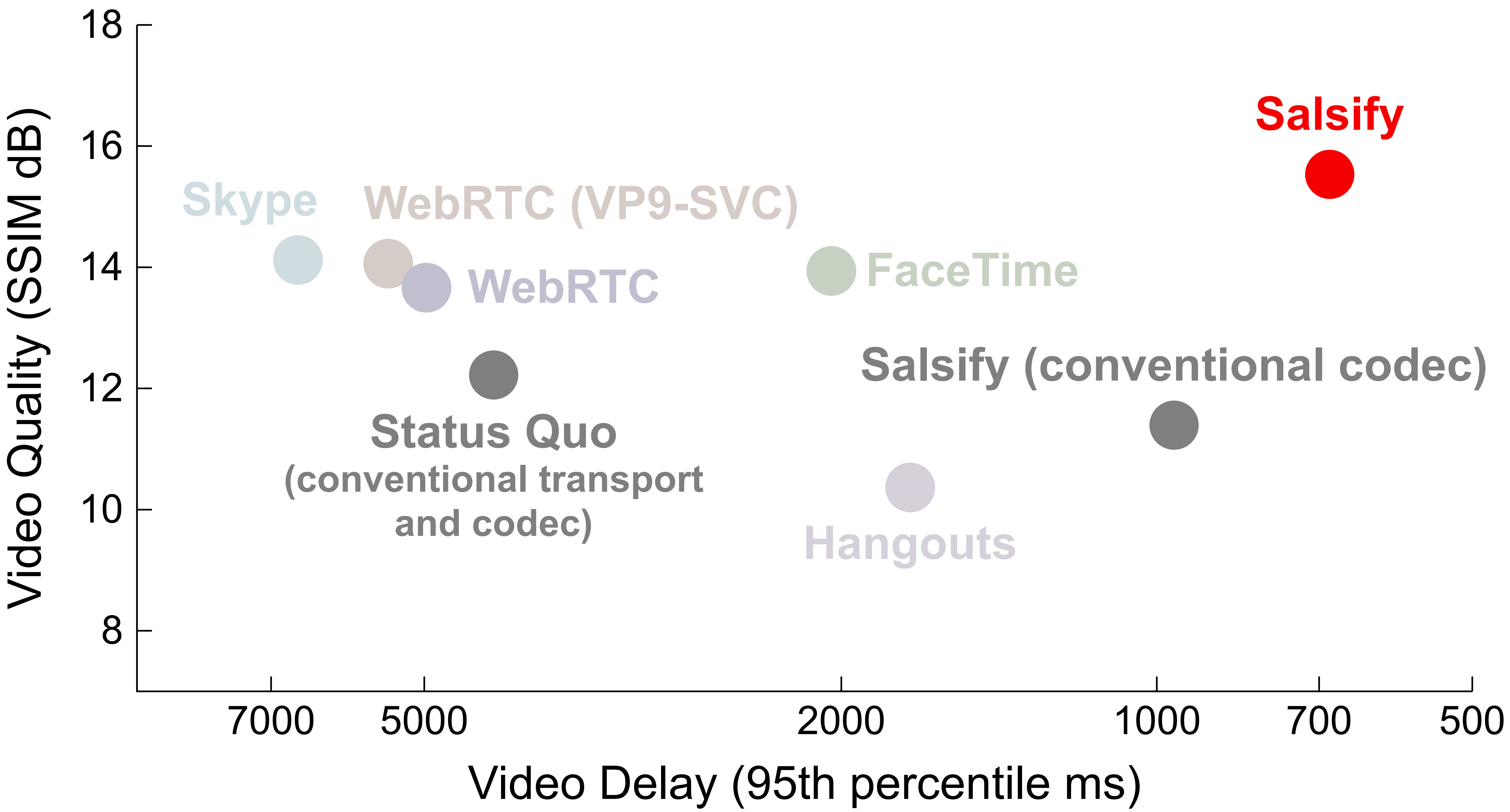
Evaluation results: **Verizon LTE Trace**



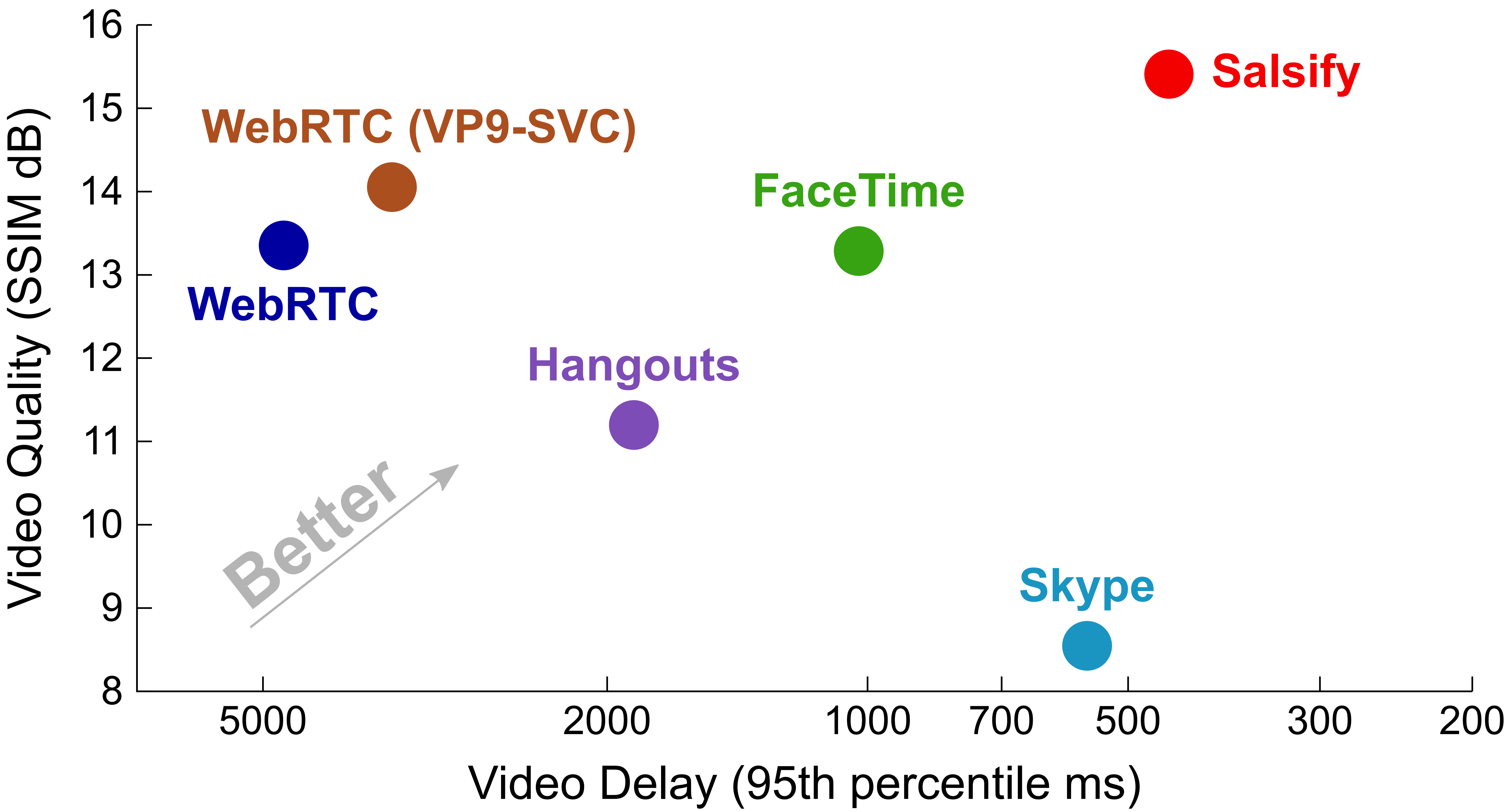
Evaluation results: **Verizon LTE Trace**



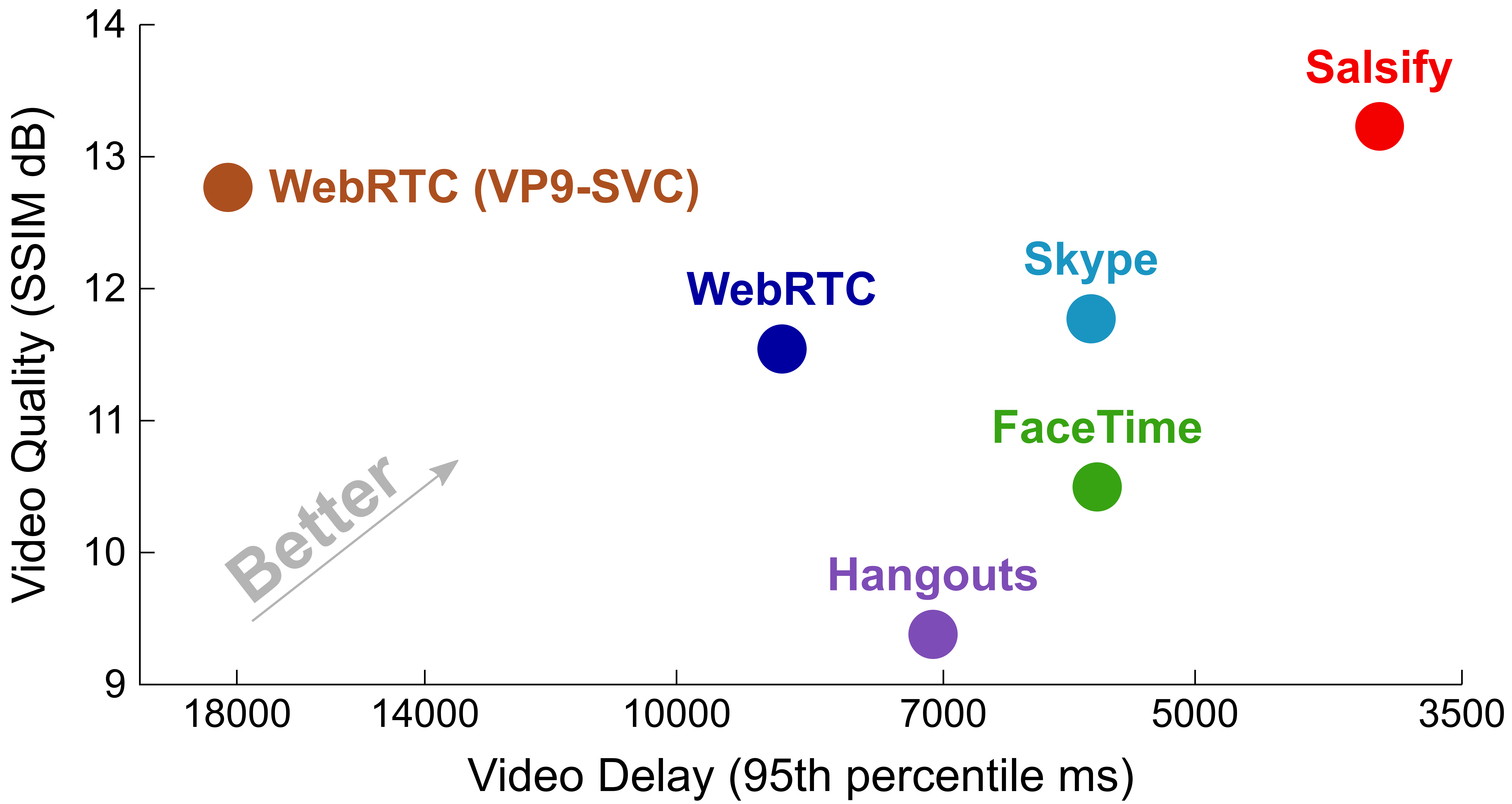
Evaluation results: **Verizon LTE Trace**

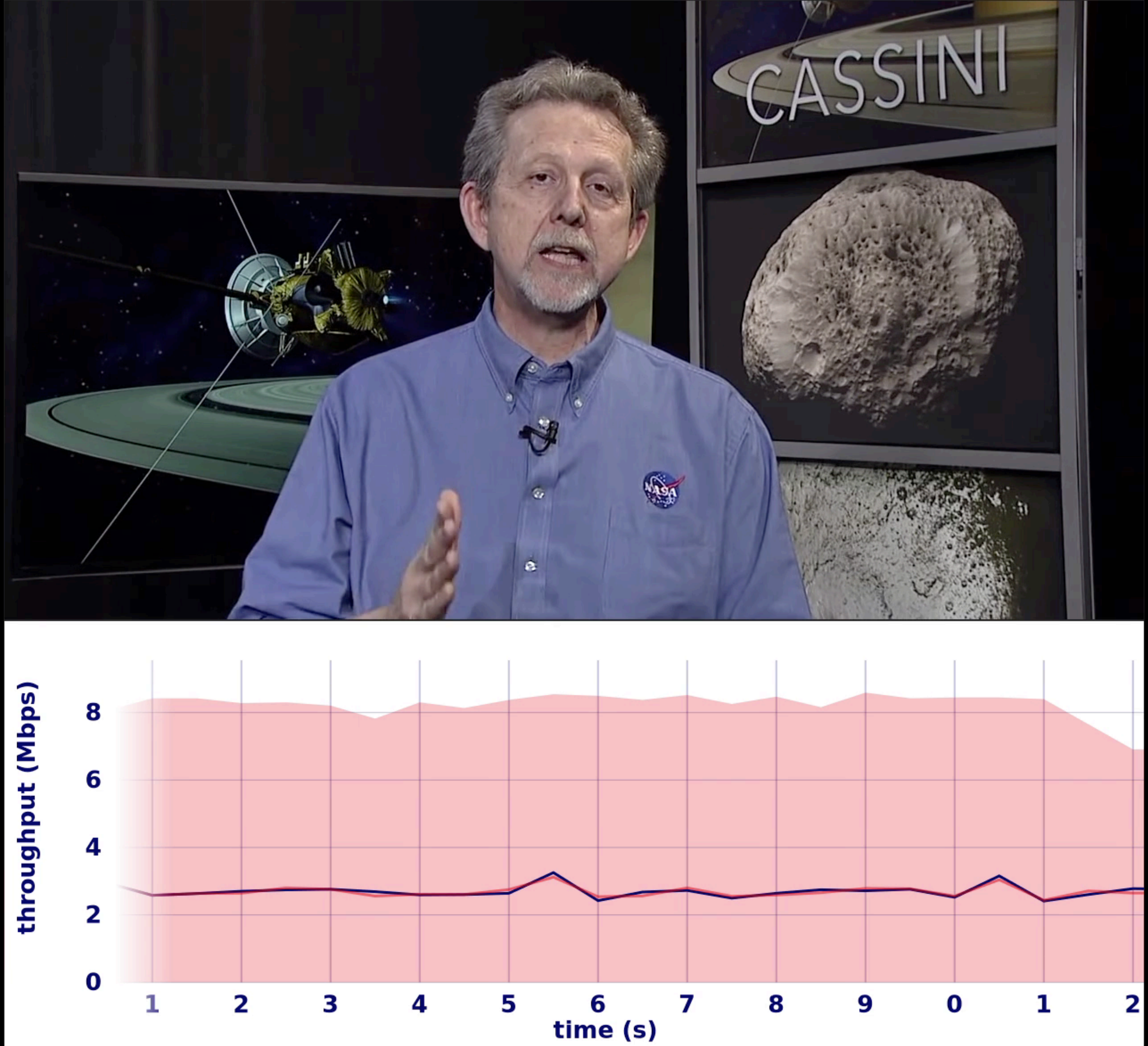


Evaluation results: **AT&T LTE Trace**



Evaluation results: **T-Mobile UMTS Trace**





WebRTC
(Chrome 65)

Improvements to ***video codecs*** may have reached the point of diminishing returns, but changes to the architecture of ***video systems*** can still yield significant benefits.

System 4: gg (laptop to lambda)

- Kalev Alpernas, Cormac Flanagan, Sadjad Fouladi, Leonid Ryzhyk, Mooly Sagiv, Thomas Schmitz, and KW, **Secure serverless computing using dynamic information flow control**, Proc. ACM Program. Lang. 2, OOPSLA, Article 118 (November 2018).
- Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and KW, **From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers**, in USENIX ATC 2019.

Cloud functions as a new computing substrate

- Rent 8,000 nodes in seconds (but some are flaky)
- Nodes can communicate directly at 600 Mbps (but some paths are flaky)
- Lots of jobs could take advantage of this substrate
  - Big compilations (compiling Chromium takes 16 hours on one core)
  - Software test suites (unit tests, integration tests)
  - Ray-tracing (rendering one frame of a movie can take >12 hours)
  - Video editing
  - Parallel jobs on large videos

The gg intermediate representation

- Types: values and thunks
- Components
  - raw inputs (“V” value name or “T” thunk name)
  - forced inputs (“T” thunk name)
  - outputs (named byte vector, may be another thunk)
  - execution spec (e.g., Unix command line)
- Addressing scheme
  - “V” + hash of a byte vector
  - **or** “T” + hash of a thunk’s canonical representation + “#” + name of an output

- Can express
  - Recursive fibonacci
  - Y combinator
  - Various everyday jobs

- Alpernas et al. (OOPSLA 2018): “Enforcing IFC policies is easy”

Compilation

① PREPROCESS(hello.c) → hello.i

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', -E', 'hello.c',
      '-o', 'hello.i' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'VLb1SuN=hello.c',
    'VDSo_TM=gcc',
    'VAs.BnH=cpp',
    'VB33fCB=/usr/stdio.h' ],
  outputs: [ 'hello.i' ] }
```

content hash: T0MEiRL

② COMPILE(hello.i) → hello.s

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'cpp-output',
      '-S', 'hello.i',
      '-o', 'hello.s' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'T0MEiRL=hello.i',
    'VDSo_TM=gcc',
    'VMRZGH1=cc1', ],
  outputs: [ 'hello.s' ] }
```

content hash: TRFSH91

③ ASSEMBLE(hello.s) → hello.o

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'assembler',
      '-c', 'hello.s',
      '-o', 'hello.o' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'TRFSH91=hello.s',
    'VDSo_TM=gcc',
    'VUn3XpT=as', ],
  outputs: [ 'hello.o' ] }
```

content hash: T42hGtG

Demo

Compiling inkscape (600 kLOC)

Tool	Time	Cost
single-core `make`	32m 34s	
"`make -j48`" on a local 48-core machine	1m 40s	
`icecc` to a warm 48-core EC2 machine	6m 51s	\$2.30/hr
`icecc` to a warm 384-core EC2 cluster	6m 57s	\$18.40/hr
gg to AWS Lambda	1m 27s	50 cents/run

Compiling Chromium (24,000 kLOC)

Tool	Time
single-core `make`	15h 58m 20s
"`make -j48`" on a local 48-core machine	38m 11s
`icecc` to a warm 48-core EC2 machine	46m 01s
`icecc` to a warm 384-core EC2 cluster	42m 18s
gg to AWS Lambda	18m 55s

Tiny functions for lots of things...

- A little "functional-ish" programming goes a long way.
- It's worth refactoring megamodules (codecs, TCP, compilers, machine learning) using ideas from functional programming.
- The ability to **name, save, and restore** program states is powerful in its own right.

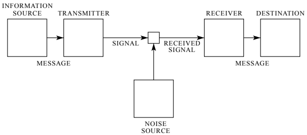
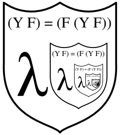


Fig. 1—Schematic diagram of a general communication system.



- **Lepton:** JPEG recompression
- **ExCamera:** video encoding with thousands of tiny tasks
- **Salsify:** real-time video with "functional" codec and transport
- **gg:** IR for "laptop to lambda" jobs with 8,000-way parallelism