Improving HTTP Latency

Venkata N. Padmanabhan *(U.C.Berkeley)*

Jeffrey C. Mogul *(DEC Western Research Lab.)*

The Second International WWW Conference, Chicago

October 17-20, 1994

Outline

- Motivation
- Sources of Latency
- Client - Server interaction
- Performance problems
- Protocol modifications
- Results
- Conclusions

Motivation

- The Web is often slow.

  Especially so when

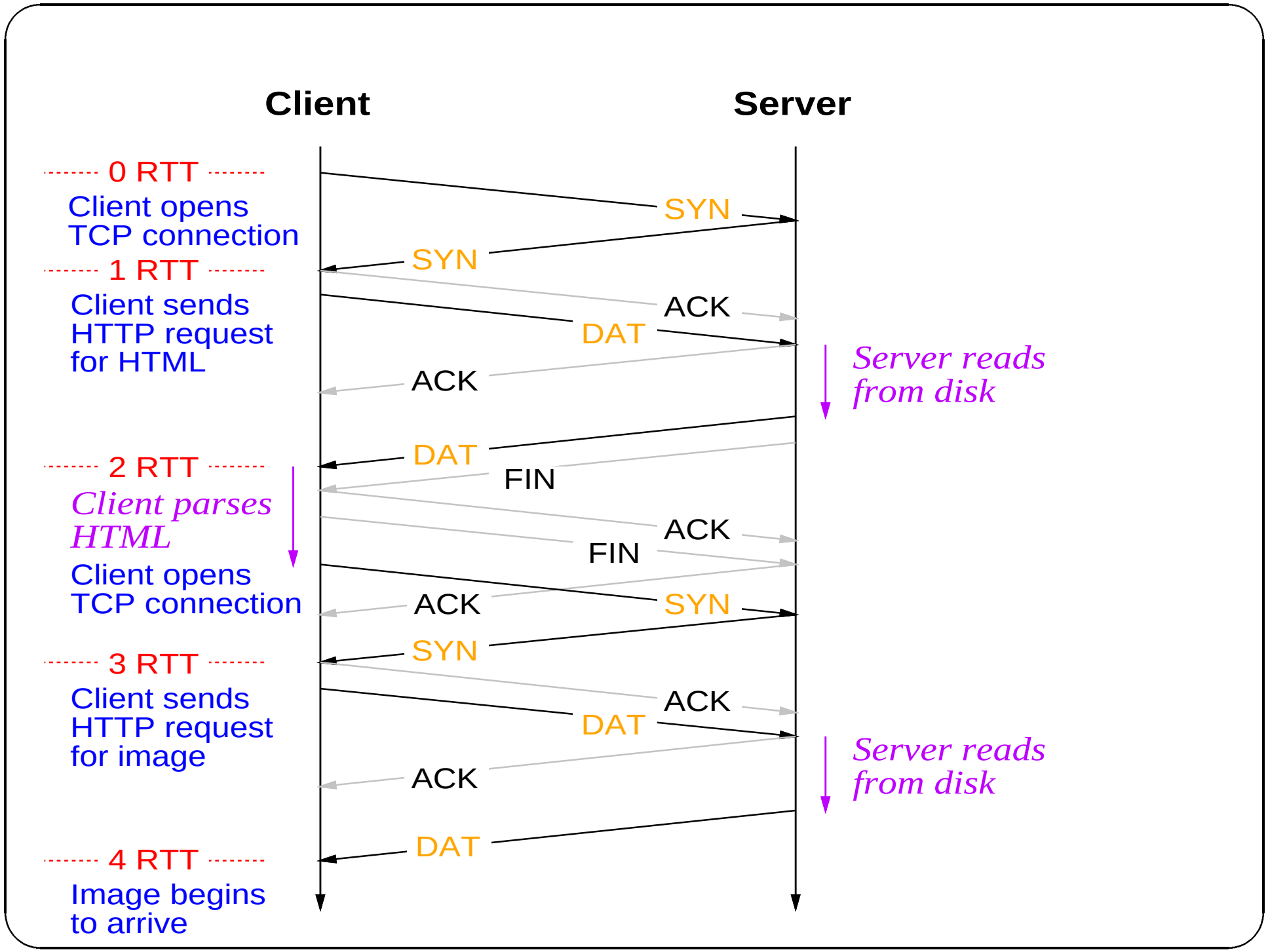
  - many inlined images
  - distant servers

- **Main Reason:** inefficient use of the network

Sources of Latency

- Server: CPU and disk speeds
  - can buy faster computers
- Client: same as above
- Network:
  - Bandwidth
    - can *possibly* buy faster links
  - Round-trip time (RTT)
    - Speed of light is a fundamental limit
    - 70 ms RTT across US, 250 ms to Australia

To reduce latency we must avoid round-trips!


Client
Server
0 RTT
Client opens TCP connection
SYN
SYN
1 RTT
Client sends HTTP request for HTML
ACK
DAT
Server reads from disk
ACK
DAT
2 RTT
Client parses HTML
FIN
ACK
FIN
Client opens TCP connection
ACK
SYN
SYN
3 RTT
Client sends HTTP request for image
ACK
DAT
Server reads from disk
ACK
DAT
4 RTT
Image begins to arrive

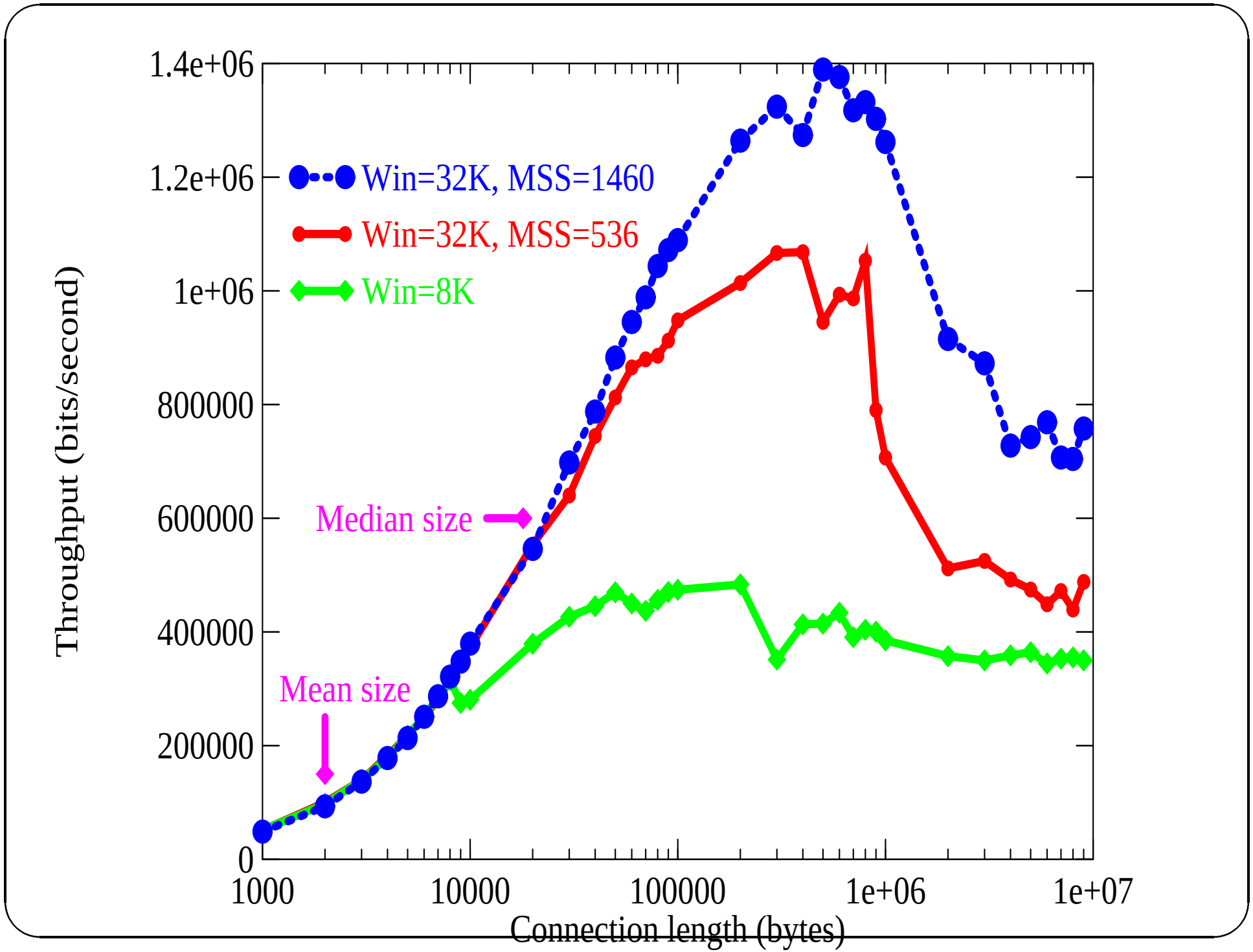
Problems

- Too many connections!
  - Processing overhead for each connection

    If authentication is done, that's extra overhead
  - 1 RTT for each connection set-up
  - TCP slowstart $\Rightarrow$ few connections reach full-steam
  - PCB table fills up with TIME-WAIT entries

  On a coast-to-coast T1 line: sending 20000 bytes achieves a throughput of only 0.6Mbps
- No pipelining

  $\Rightarrow$ each inlined image requires atleast an additional roundtrip


1.4e+06
1.2e+06
1e+06
800000
600000
400000
200000
0
Throughput (bits/second)
Win=32K, MSS=1460
Win=32K, MSS=536
Win=8K
Median size
Mean size
1000
10000
100000
1e+06
1e+07
Connection length (bytes)

[7]

Persistent Connections

- Client tells server to keep connection open
  - uses HTRQ (HT Request) headers, so interoperates fully
  - future versions of HTTP can define a `hold-connection` pragma
- server process loops waiting for requests
- client or server can close connections to conserve resources

Marking the end of data

HTTP gives server 3 ways to mark end of data:

- Content-length field
- Content-type field (MIME delimiter)
- Close connection
  - current implementations do this

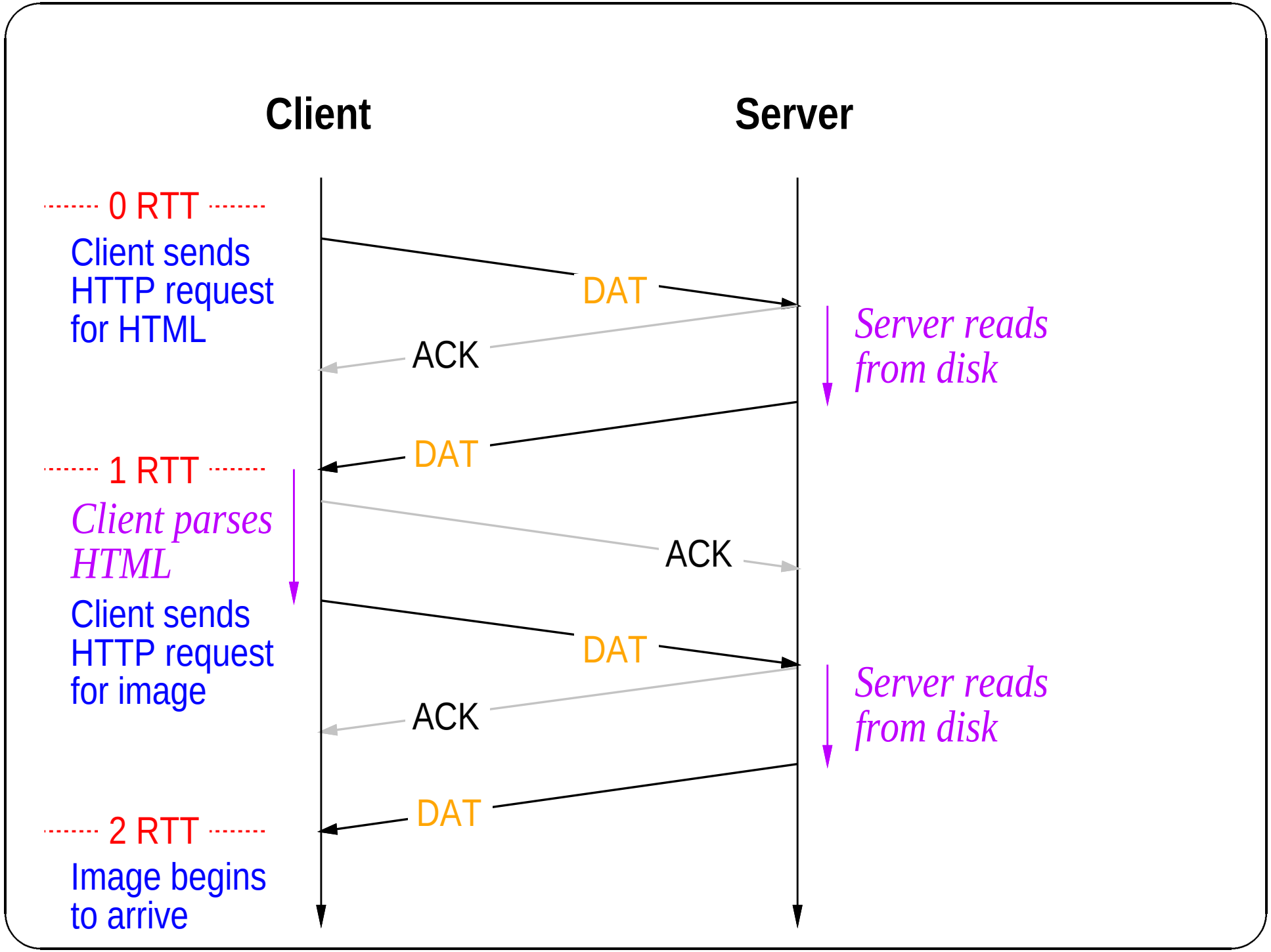
We chose the first alternative.

When data comes from a subprocess (script)

- Server doesn't know where data ends
- Simple way out: close the connection

Some alternatives:

- a separate control connection
- block-by-block transfer


Client
Server
0 RTT
Client sends HTTP request for HTML
DAT
ACK
Server reads from disk
DAT
1 RTT
Client parses HTML
ACK
Client sends HTTP request for image
DAT
ACK
Server reads from disk
DAT
2 RTT
Image begins to arrive

Pipelining requests

Even with persistent connections, still takes 1 RTT per inlined image

Client *knows* what images are needed after parsing HTML

- could request all needed images at once

  $\Rightarrow$ Best case: 2 RTTs per document

Server *could know* what images are needed when HTML request arrives

- could return all images immediately

  $\Rightarrow$ Best case: 1 RTT per document

GETALL

- GET $< HTML\_document >$

  $\Rightarrow$ server sends back only the document

We define:

- GETALL $< HTML\_document >$

  Server sends back document and all inlined images
- can be implemented as a GET with a pragma

Potential problems:

- Server has to parse the HTML.
  - can be done once, and the results cached
- Server could return image already cached by client

GETLIST

Another primitive:

- GETLIST $< URL\_list >$

  $\Rightarrow$ Server sends back all the requested documents

- can be implemented as a bunch of GETs

Overall scheme:

The client

- uses GETALL for the first access
- keeps cache of image URLs of recently accessed pages
- uses GETLIST for subsequent accesses to request only images required

Results: Remote server

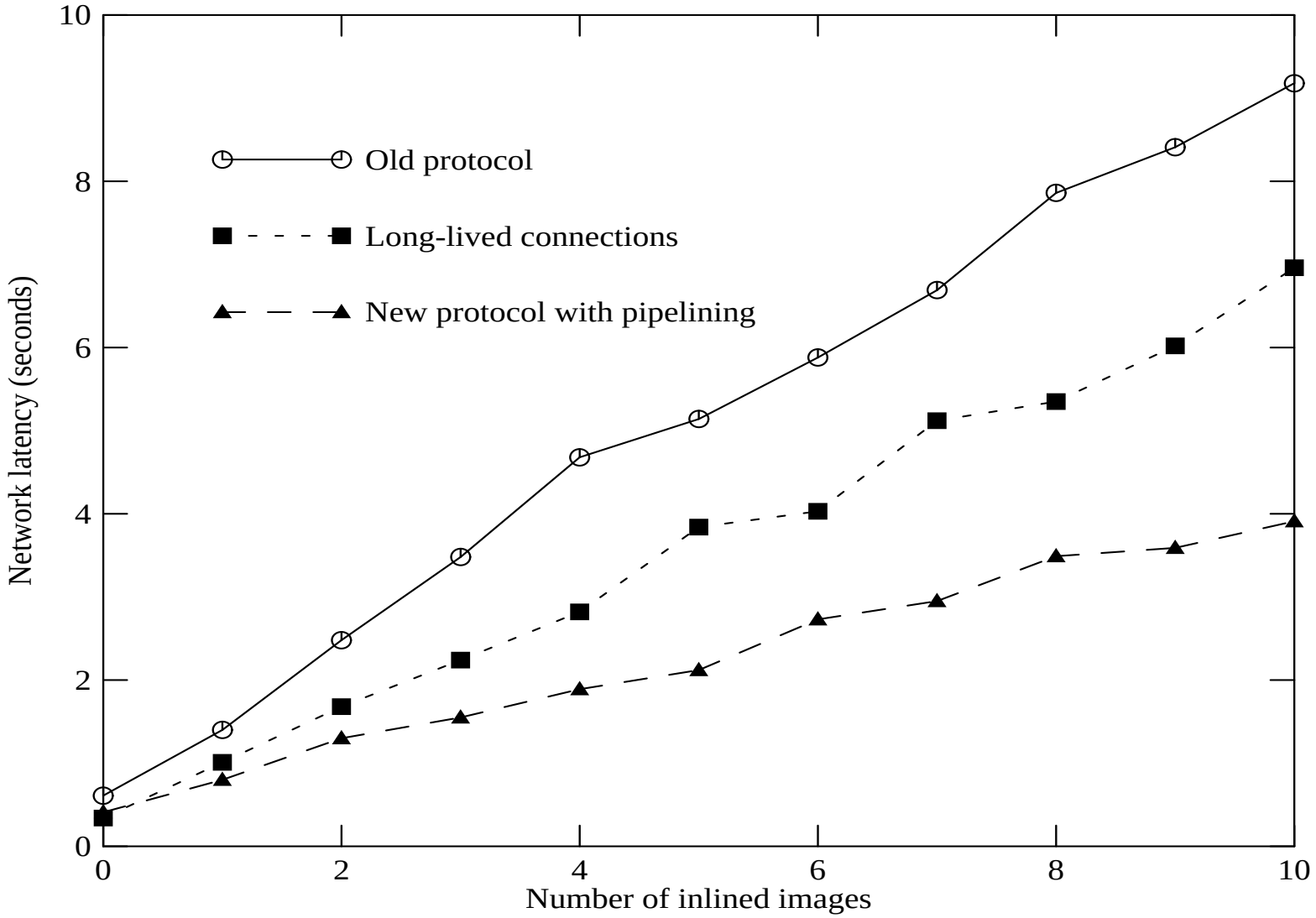


Image size: 2544 bytes

Results: Remote server

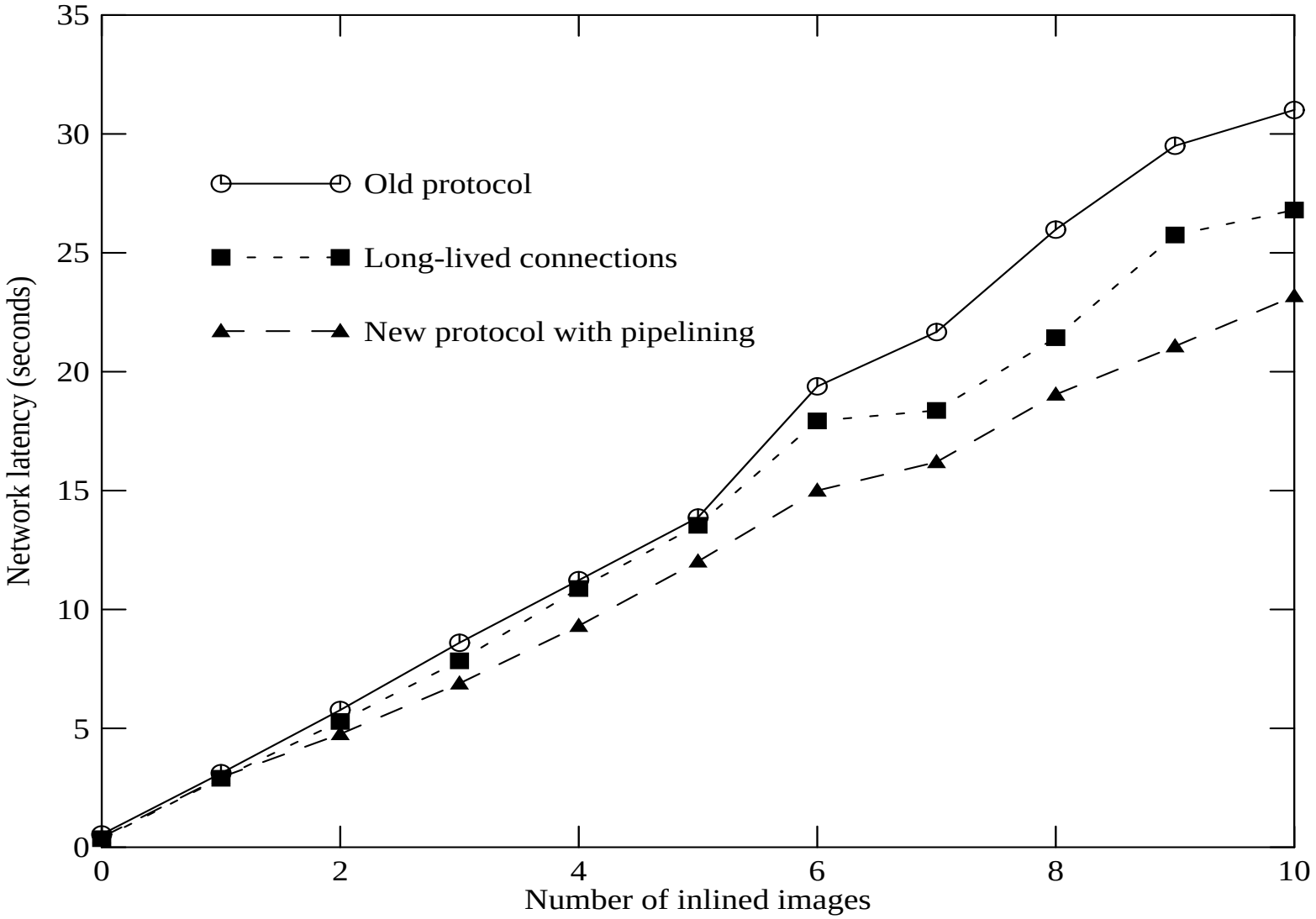
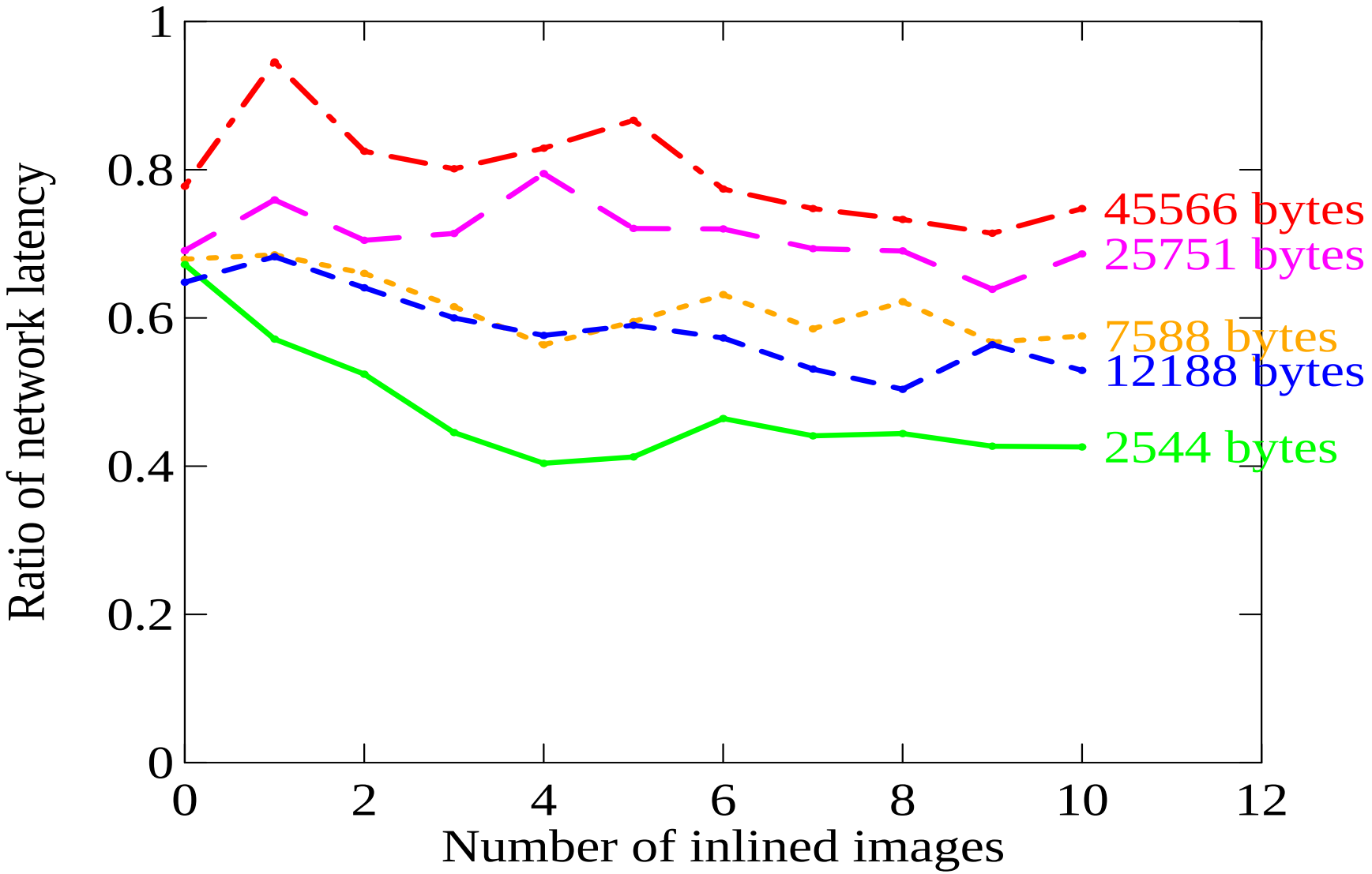
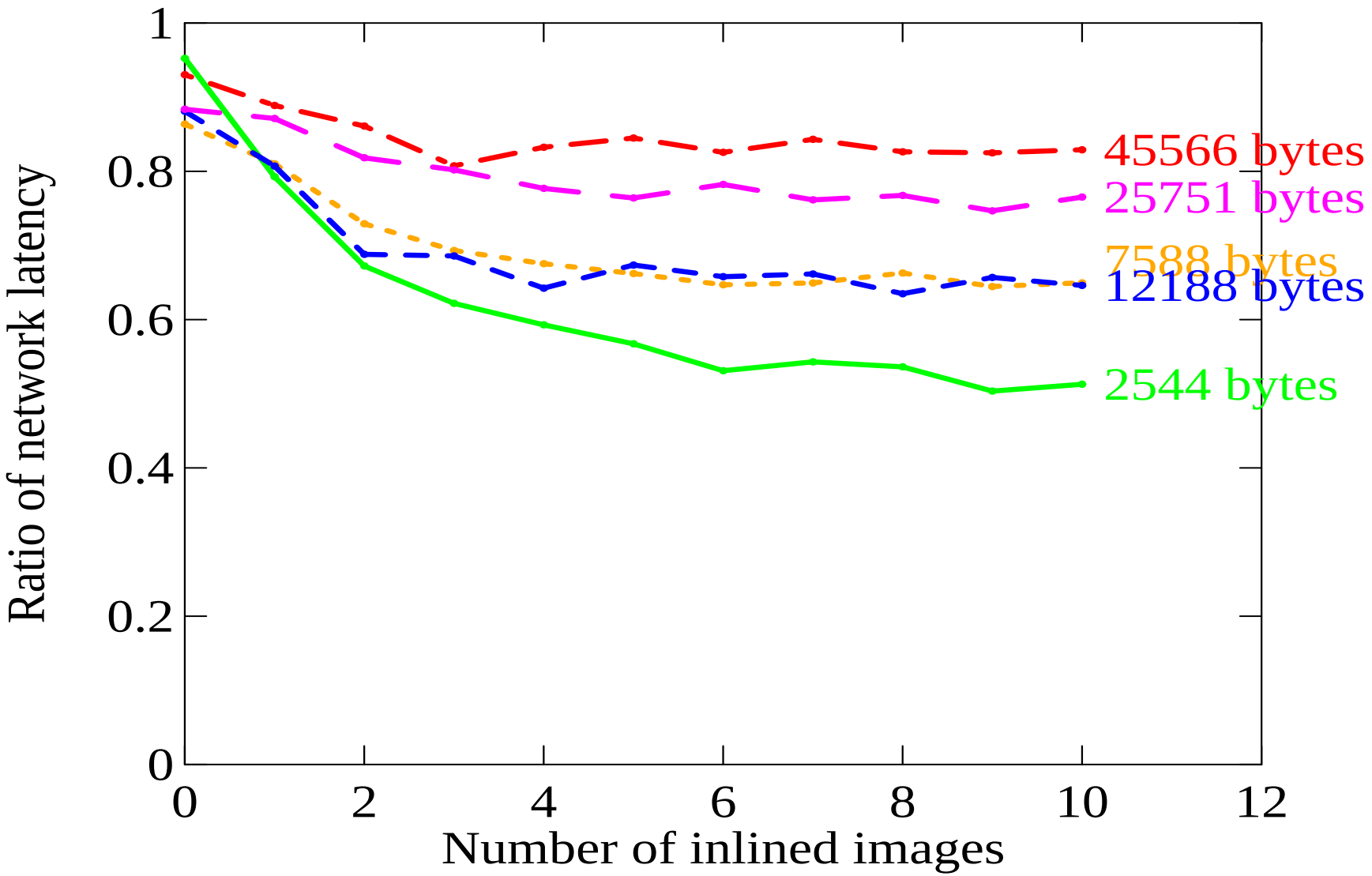


Image size: 45566 bytes

Summary: Remote server



Summary: Local server



Conclusions

- With a slightly modified protocol, there is a substantial reduction in latency
- Improvement depends on size and number of images
  - 20-60% for remote server
  - 15-50% for local server
- Full interoperability