CombiHeader: Minimizing the Number of Shim Headers in Redundancy Elimination Systems

Sumanta Saha,
Andrey Lukyanenko and
Antti Ylä-Jääski
Aalto University School of Science, Finland
(Formerly, Helsinki University of Technology)
Outline

- Redundancy elimination systems
- Finer vs. coarser chunk size
- CombiHeader algorithm
- Proof-of-Concept implementation
- Evaluation
- Summary
Redundancy Elimination Systems

- Redundancy Elimination (RE) systems work on packet payload level
  - Chunks the payload using Rabin Fingerprinting
  - Content based
  - Application independent

- The idea is complementary to traditional caching
  - Aims to remove redundant content from upstream nodes to downstream

- Eliminates duplicate traffic when traditional caching fails
Redundancy Elimination Systems
Finer vs. Coarser Chunk Sizes

- With coarser chunk size there is always a possibility of missing possible matching regions
- Finer chunk sizes have more protocol overhead
- The proposed algorithm, CombiHeader, uses an adaptive method to dynamically choose the best chunk size
- Please refer to the paper for a mathematical interpretation
CombiHeader

- We need a dynamic system to adapt to the content type, and chunk popularity to get the best out of it
- CombiHeader works on chunk popularity to generate bigger chunks out of smaller ones
- Optimized to deliver least memory access while matching to the largest chunk possible

Algorithm 1 Pseudocode of CombiHeader algorithm

```plaintext
if current chunk is cache miss then
  if miss streak = 1 then
    Update last seen Combi and Elementary node
    Create new complex CombiNode if possible
  else
    Clear last CombiNode and transmit it
    Update last seen elementary node
  end if
  Transmit full text of current chunk
else
  If miss streak = 1 then
    Start fresh, clear last seen CombiNode
    Do not transmit anything
    Buffer current chunk as last seen elementary
    miss streak ← 0
  else
    Update last seen elementary node
    Try to create new CombiNode with (Last CombiNode, current Node)
    if Try failed then
      Transmit and clear last CombiNode
    else
      Do not transmit anything
      Complex CombiNode is accumulating chunks to transmit
    end if
  end if
end if
```
Chunk Trail: H1H2H3H4H1H2H3H1H2H3
CombiHeader

Insertion of CombiHeaders to the outgoing stream

Trail: h1h2h3h4h1h2h3h1h2h3h5
Last elementary: h1  h2  h3  h4  h1  h2  h3  h1  h2  h3  h5
Last CombiNode : -  -  -  -  -  h1h2  -  -  h1h2  h1h2h3  -
Cache hit/miss : M  M  M  M  H  H  H  H  H  H  M
Insert in trans: F(h1) F(h2) F(h3) F(h4)  -  -  h1h2  h3  -  -  h1h2h3
& F(h5)

F(h_x)  = Full payload for chunk x
h_x    = Elementary header for chunk x
h_xh_y = CombiHeader for combined chunks x and y
Proof-of-Concept

- Implementation done in pure C
  - Chunking engine
  - CombiHeader plug-in
- Rabin fingerprinting for RE
- SHA-1 hashing for fingerprinting
- Chunking can be done in both IP and TCP layer
  - Experiments were done on TCP layer
- Directed graph to keep track of all the CombiHeaders generated
- A threshold parameter $\theta$ is used to control the CombiHeader generation process
Evaluation

- Effect of CombiHeader over header transmission
  - X-axis represents initial preliminary chunk size
  - Traffic comprises of video files with intermittent similarity

- CombiHeader allowing smaller chunk size with the same benefit as larger ones
Evaluation

- Effect of CombiHeader over total bytes transmitted to wire
  - X-axis represents the number of files transferred through the router

- Running CombiHeader on real world HTTP traces
Summary

- CombiHeader addresses the question of what should be the optimal chunk size for a particular traffic
- Depending on the dynamic nature of the user traffic and the underlying similarity, CombiHeader adapts itself to deliver the best possible chunk size
- Helps to reduce protocol overhead to the wire
- Possible deployment challenges:
  - Cache synchronization among routers
  - Routing decision making