

# Varnish http accelerator

---

- A story about 2006 software design

Poul-Henning Kamp

[phk@FreeBSD.org](mailto:phk@FreeBSD.org)

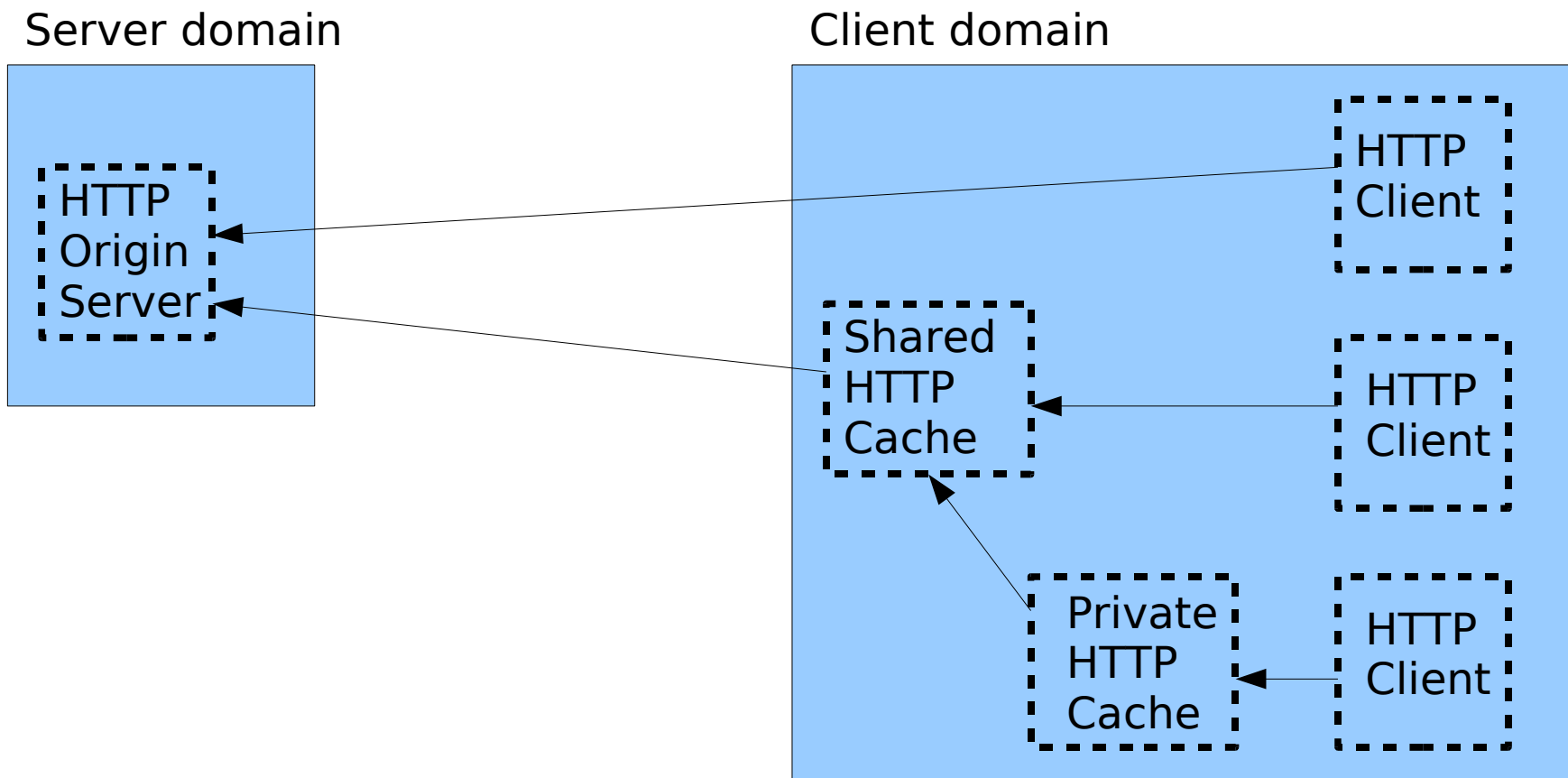
# Why I wrote Varnish

---

- I was sceptical when VG approached me with the Varnish Project.
  - Not really my specialty
  - Not really my preferred area
- On the other hand
  - I'm tired of people writing lousy programs and blaming it on "my" kernel.
  - Good chance to educate by example.

# RFC2616 on HTTP caches

---



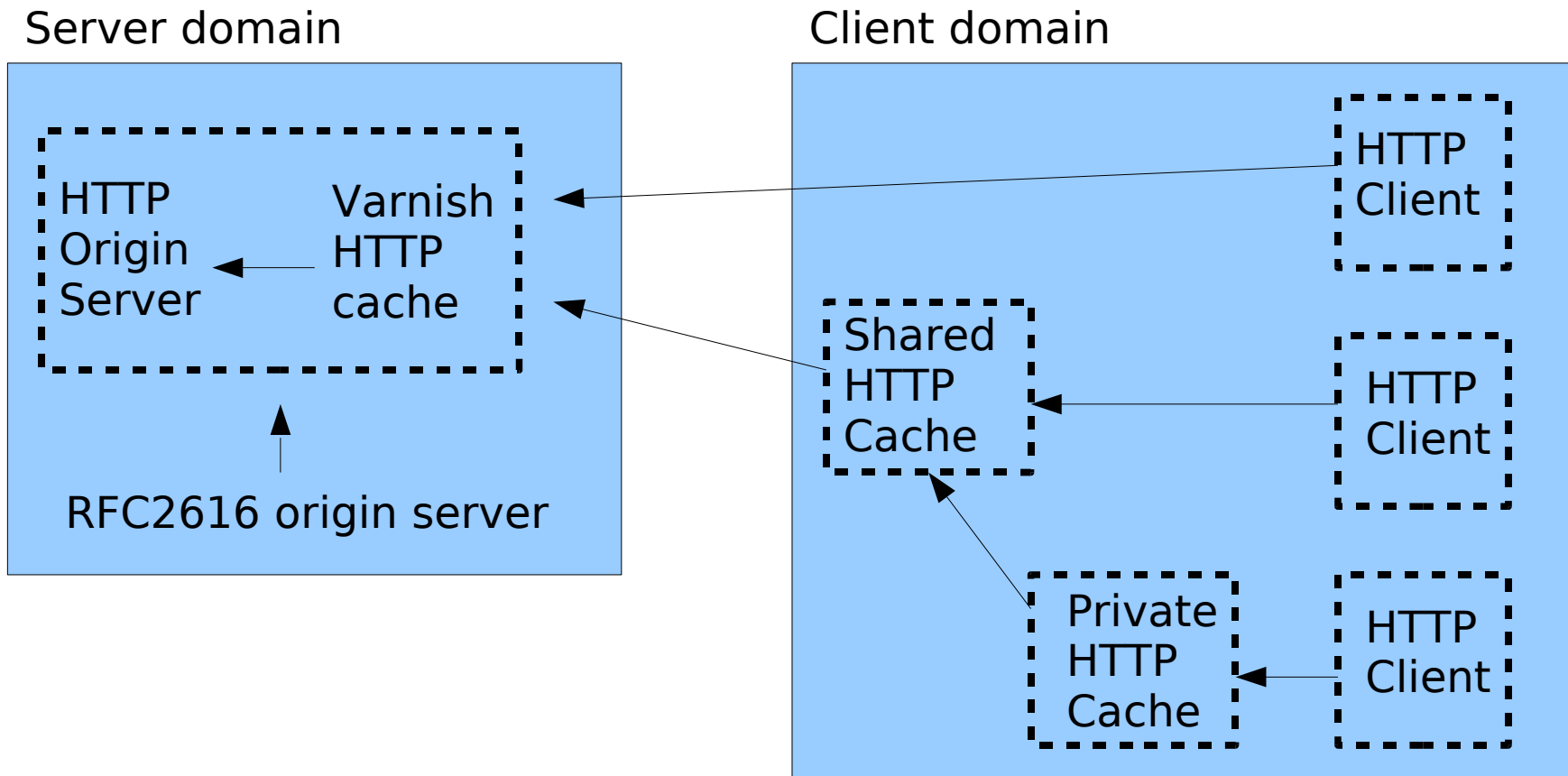
# Client Cache Situation

---

- Origin servers are adversarial.
- Anything the origin server says is law
  - ... if we can make sense of it.
- If in doubt: don't cache.
- Be semantically transparent at any cost.
- If origin server does not reply: error.

# RFC2616 and Varnish

---

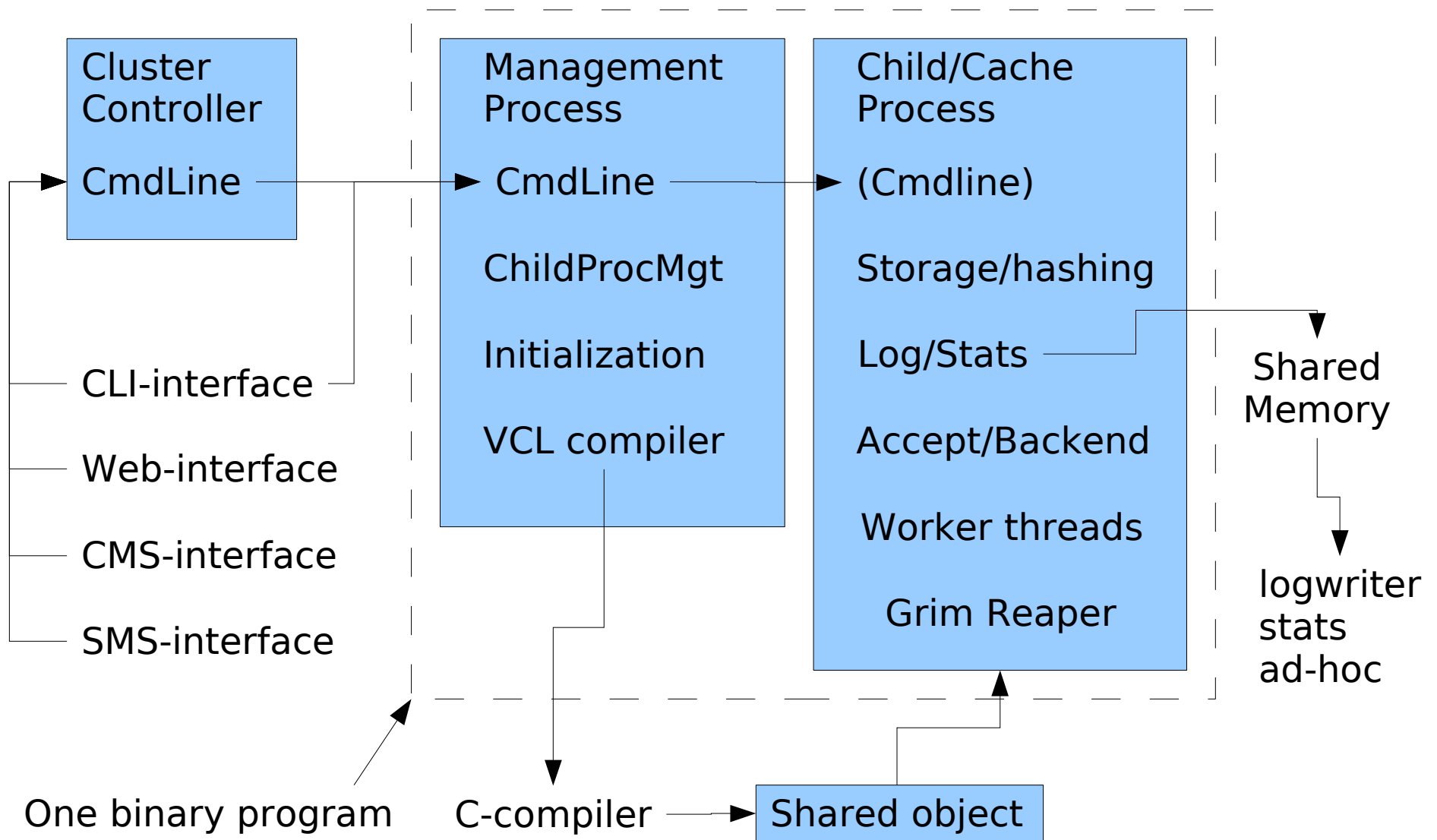


# Server Cache Situation

---

- Backend (origin server) is on our side.
- We might be responsible for modifying the origin servers instructions.
  - Change TTL, rewrite URLs etc.
- Whatever happens: protect the backend.
- If backend does not reply: do something!

# Varnish Architecture



# Varnish Config Language

---

- Simple domain specific language
  - Compiled via C language to binary
    - Transparantly!
  - Dynamically loaded
  - Multiple configs loaded concurrently
- Instant switch from one VCL to another.
  - Can be done from VCL(!)

# VCL example

---

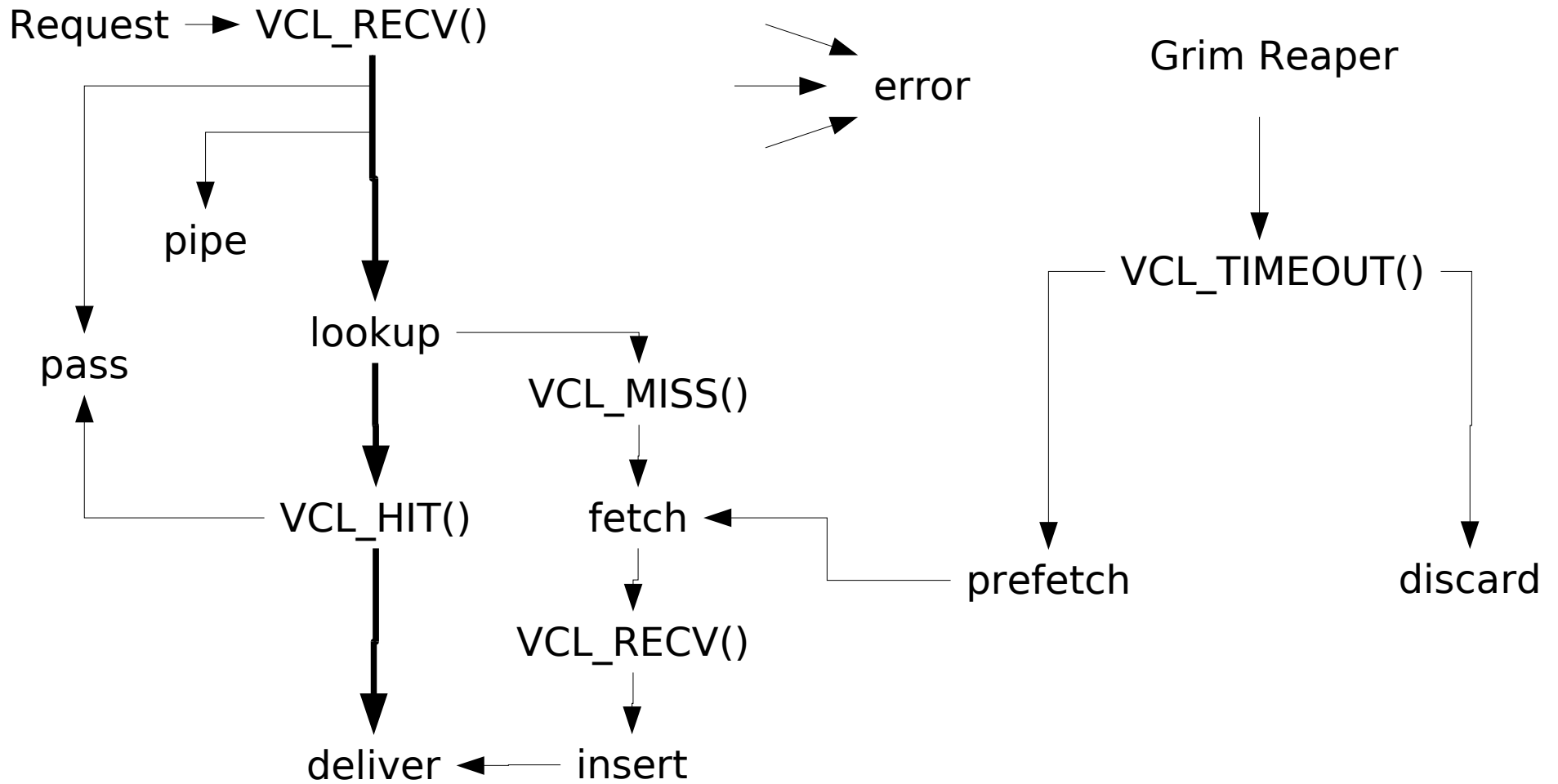
```
if (client.ip in 10.0.0.0/8) {
    pass;
}

if (req.url.host ~ "cnn.no$") {
    rewrite req.url.host "cnn.no" "vg.no"
}

if (!backend.up) {
    if (obj.exist) {
        set obj.ttl += 10m;
        deliver;
    }
    switch_config "ohhshit";
}
```

# VCL structure

---



# Performance Design

---

- Don't copy data if we can avoid it
  - Avoid text-processing headers
- Convert Chunked encoding to Direct
- Also cache "cannot be cached" info
- Maximize session usage
  - Pass-through mode understands chunked encoding etc.
  - Pipe mode for weird stuff (selected by VCL)

# Performance Programming

---

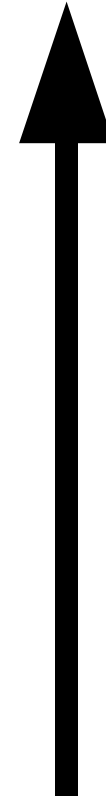
- Use few cheap operations
- Use even fewer expensive operations.
- Don't fight the kernel.
- Use the features the nice kernel guys have given you.

# Price List

---

- Filesystem operation
- Disk access
- Context switch
- System Call
- Locking
- Memory copy
- `Strlen(p);`
- `char *p += 5;`

Expensive



Cheap

# Logging

---

- Traditional logging is expensive.

```
FILE *flog;

flog = fopen("/var/log/mylog", "a");
[...]
fprintf(flog, "%s Something went wrong with %s\n",
        timestamp(), myrepresentation(object));
fflush(flog);
```

# Cheap logging

---

- Shared memory is perfect for cheap and fast logging.

```
Char *logp, *loge;

logp = mmap(..., size);
loge = logp + size;
[...]
logp[1] = LOG_ERROR;
logp[2] = sprintf(logp + 3,
    "Something went bad with %s",
    myrepresentation(obj));
logp[3 + logp[2]] = LOG_END;
logp[0] = LOG_ENTRY;
logp += 3 + logp[2];
```

# But it can be even cheaper

---

- Keep track of both ends of strings.
- Use knowledge of length where possible.
  - Use `memcpy(3)` instead of `strcpy(3)`
- ```
void VSLR(enum shmlogtag tag,  
          unsigned id,  
          const char *b, const char *e);
```
- ```
void VSL(enum shmlogtag tag,  
         unsigned id,  
         const char *fmt, ...);
```

# But it can be **even** cheaper

---

- Add a shmlog buffer to worker thread
- Logging to this buffer is lockless
- Batch "commit" local buffer to "real" shmlog at synchronization points.
- => Practically no mutex contention.

# Varnish logging

---

- Logging to shared memory
  - No slowdown for the real workload.
- Daemons tailing shm can cheaply generate the "real" output:
  - Apache format
  - Custom format
  - Realtime views

# ...and

---

- Coredumps contains the most recent log records.
- We can afford to log much more detail, it does not take up disk-space.
- Approx 30 records per request
  - =>100.000's records/sec

# “varnishtop”

---

- Example of ad-hoc shmlog reader:
- `./varnishtop -i sessionclose`

- Why do sessions close ?

```
132232.12 timeout
39002.59 EOF
5232.97 not HTTP/1.1
3069.28 Connection: close
1383.40 remote closed
630.93 silent
15.49 pipe
9.15 Not in cache.
6.36 Purged.
5.41 no request
0.98 Bad Request
```

# Statistics

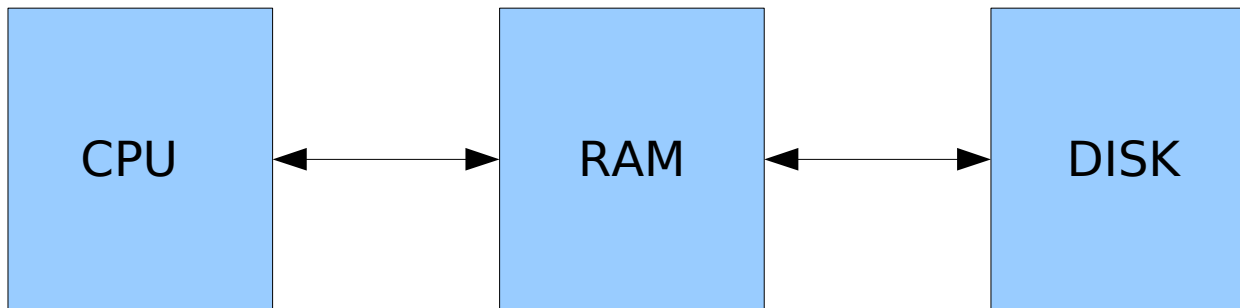
---

- Same story.
- Stored in shared memory
- Programs can monitor & present data
- No system calls necessary to get up to date numbers.

# People program like 1970

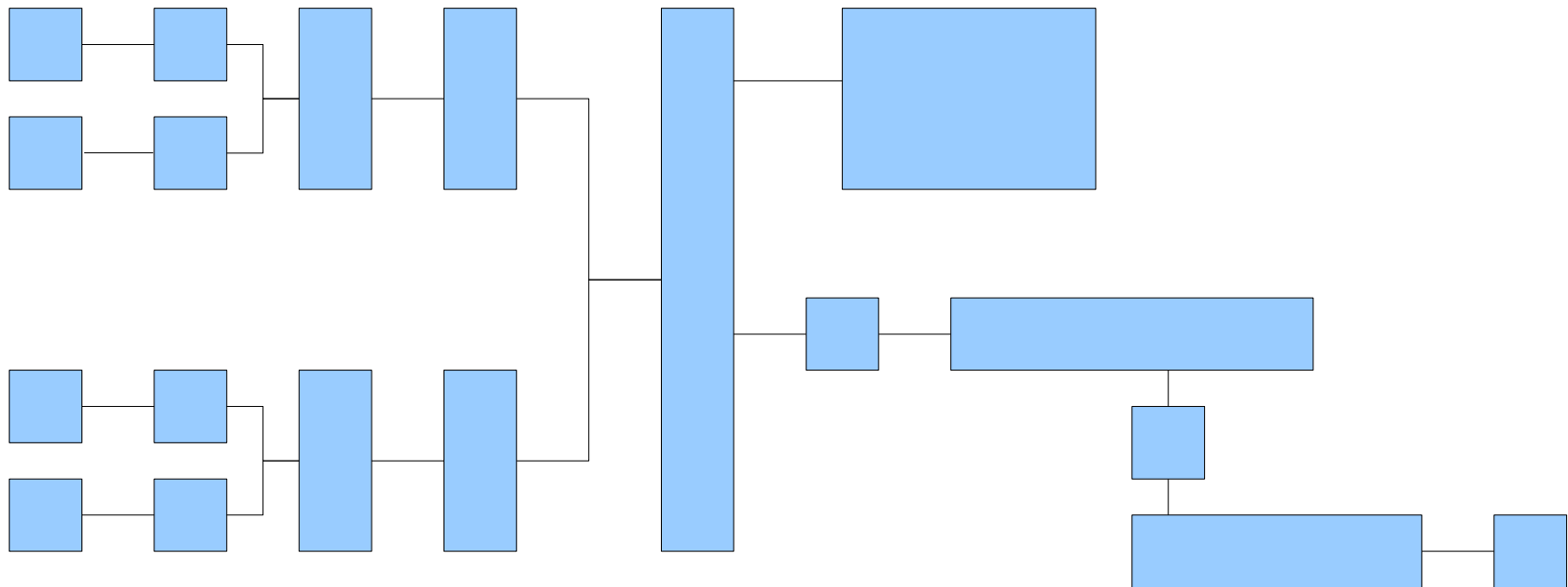
---

- "A computer consists of a CPU, internal storage and external storage"



# 2006 reality

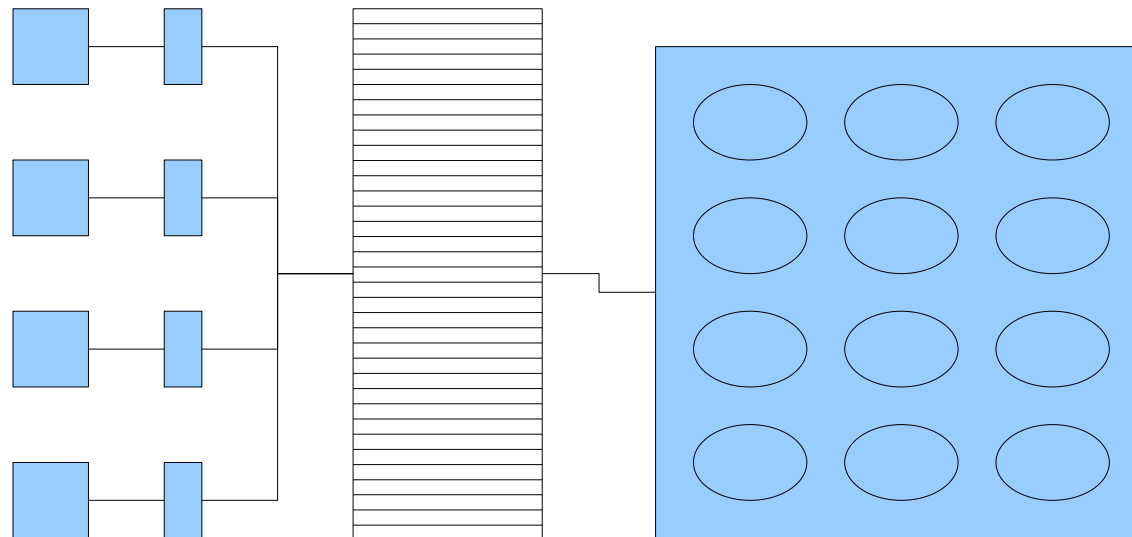
- "A Computer consists of a number of processing units ("cores") connected via a number of caches and busses to an virtualized storage."



# 2006 conceptual model

---

- "A computer consists of a number of cores with private caches, a shared (page granularity) cache for external storage objects."



# Squid is 1970 model

---

- Some objects "in Memory"
- Other objects "on disk"
- Explicit transfer between the two states:
  - Memory object unused gets written to disk
  - Disk object gets read to memory for transmission.

# Fight the kernel (1)

---

- Squid creates a memory object.
- Gets used some then falls into disuse.
- Kernel pages object out, needs RAM.
- Squid decides object is unused and should be moved to disk.
- Creates file, issues write(2)
- Kernel takes page-fault on non-resident object.

# Fight the kernel (2)

---

- Kernel reads pages from disk.
- Squids write(2) writes object to disk
- Squid can reuse memory for other object.
- Squid needs object back
- Squid repeats all of above with some other object, then read(2) to get object from disk into newly freed storage.

# Varnish

---

- Varnish creates object in virtual memory.
- Object used some, then falls into disuse.
- Kernel pages object out to free RAM for better use.
- Varnish access object in virtual memory
- Kernel page-faults, reads object from disk.

# Virtual memory shortage

---

- On 32 bit systems VM is limited to ~2GB
- If you have a webserver with a larger working set of content, you can afford to buy a new 64 bit CPU.

# Text processing is bad.

---

- Don't waste time on text-processing.
- Don't parse fields until you need them.
- Receive http header into buffer.
- Split complete header into fields
  - Keep track of start/stop, insert NUL for all fields.

# How to find a http header

---

- Encode length in literal strings:

- "\007Expect:"

```
static unsigned
http_findhdr(struct http *hp, unsigned l, const char *hdr)
{
    unsigned u;

    for (u = HTTP_HDR_FIRST; u < hp->nhd; u++) {
        if (hp->hd[u].e < hp->hd[u].b + l + 1)
            continue;
        if (hp->hd[u].b[l] != ':')
            continue;
        if (strncasecmp(hdr, hp->hd[u].b, l))
            continue;
        return (u);
    }
    return (0);
}
```

# Memory management

---

- During the handling of a request we need various bits of memory.
- Use a private pool.
- Free everything with a single pointer assignment at the end.
- Preallocate objects before grabbing locks.

# Remember the caches

---

- Reuse most recently used resource.
  - It's more likely to be in cache.
- Process, Thread, Memory, File, Directory, Disk-block etc.
- Avoid round-robin scheduling.
  - LIFO, not FIFO.