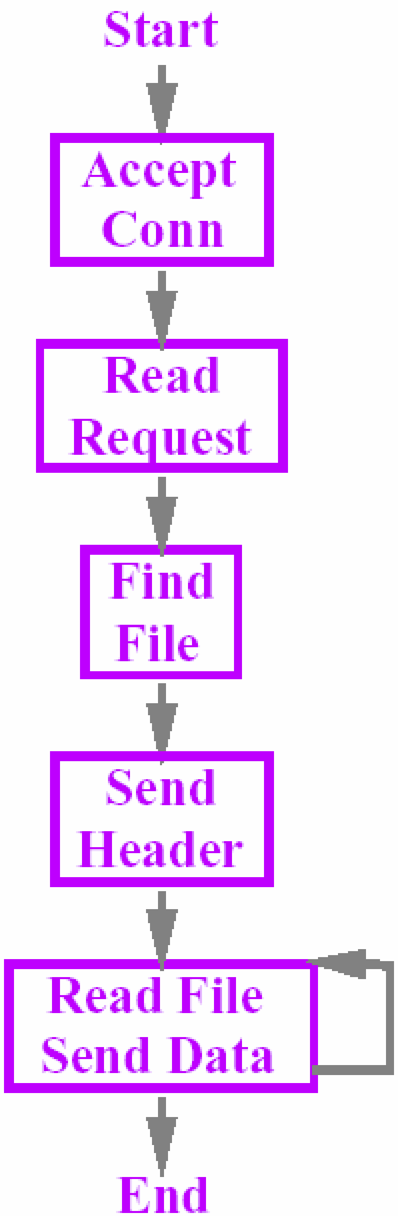
Webserver architecture

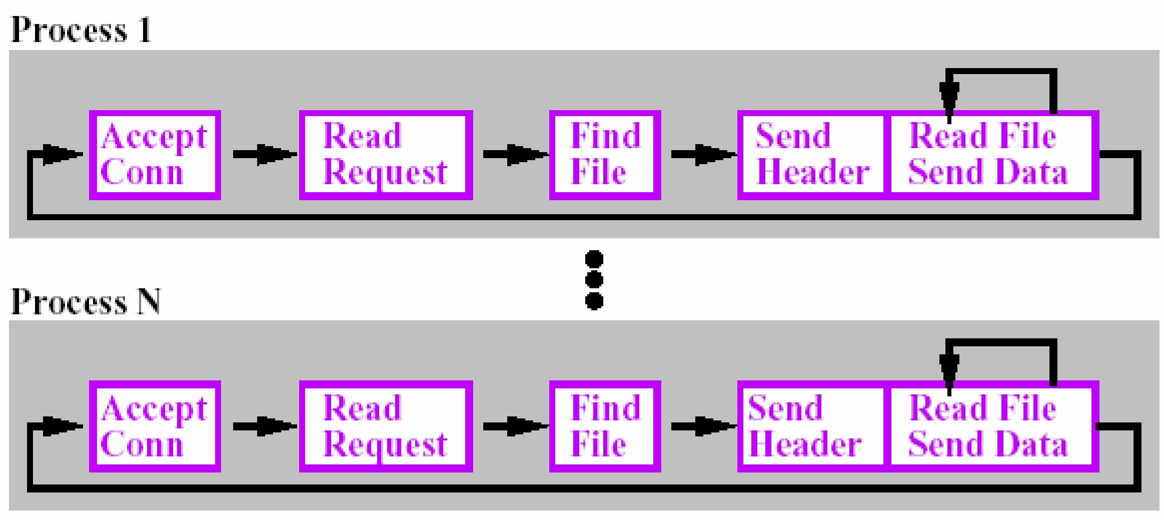
Discussion based on paper “Flash: An efficient and portable Web server”



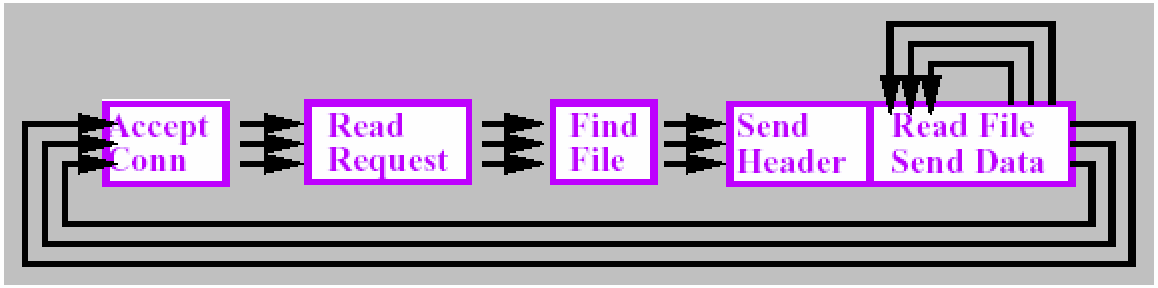
- basic processing steps
- can block in accept, read_request, send_data on network; find or read file on disk IO
- want architecture that's portable,
  - i.e. if OS doesn't support threads, or non-blocking calls, want still the same API
- want to improve performance
  - avoid cost of context switch if possible
  - want to exploit cache locality

Web server architectures

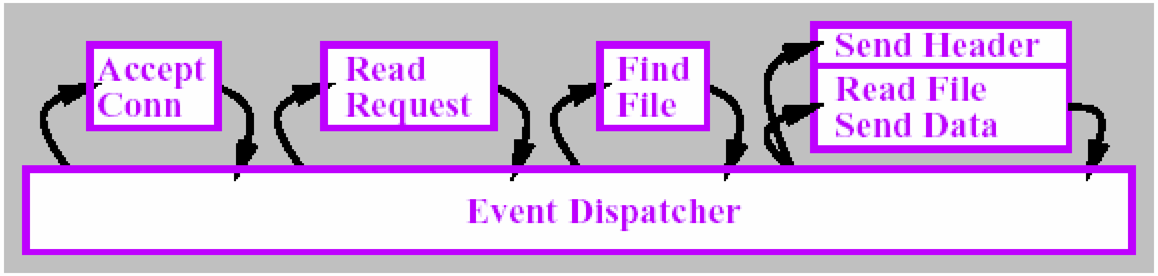
- Multi-process
  - simple programming,
  - many processes, bad memory utilization, caching and exchange of information harder



- Multi-threaded
  - cheaper context switch, shared address space
  - requires synchronization, underlying support for threads



- Single Process Event Driven (SPED)
  - single address space
  - single thread of control, no synchronization
  - use of select to wait on network events
  - in practice still blocks on disk reads

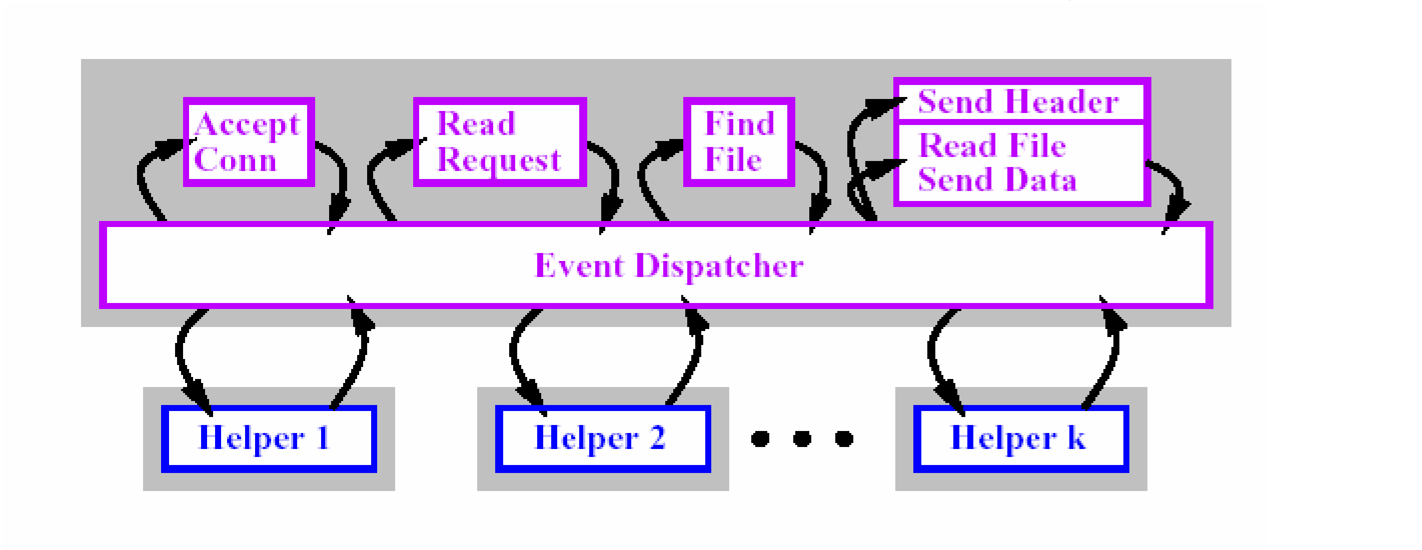


… back to paper

- problem with SPED model
  - server performs `select()`, and based on completed I/O initiated the next basic step
  - if step completes immediately, good, if it needs to block, non-blocking calls are used in SPED model, call returns to server, and SPED will figure out it has completed when it sees it on future `select()`
  - however, in some s-ms, asynchronous calls for disk I/O have alternate APIs, and cannot be integrated with the `select()` used by SPED server!

They propose – Asymmetric Multi-Process Event-Driven model

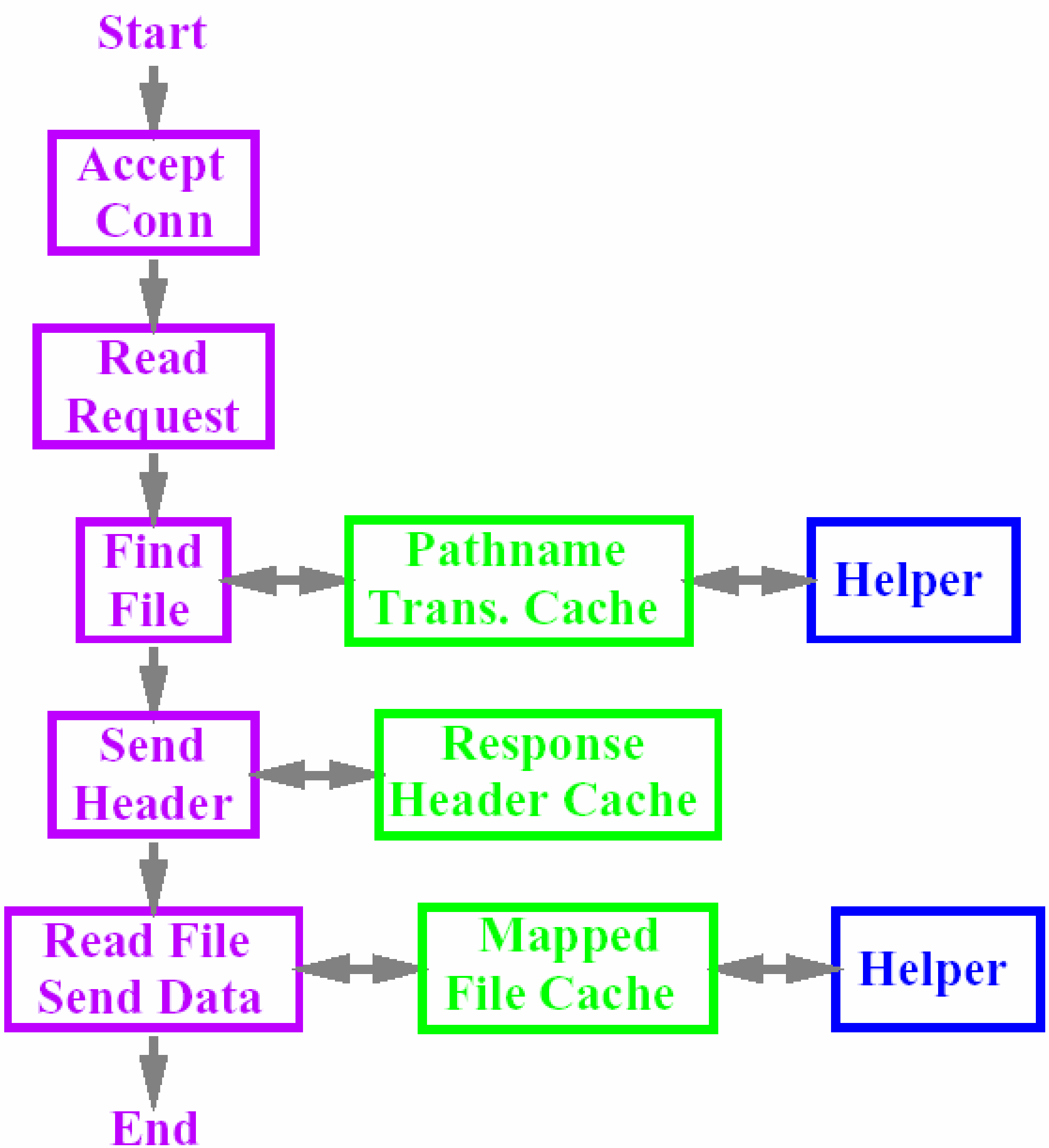
- Propose model – implement Flash
- Helper threads/processes communicate with pipe or socket IPC with the main server just to accept work requests and notify of their completion – and can be integrated with select()!!!
- The actual data is passed in shared memory

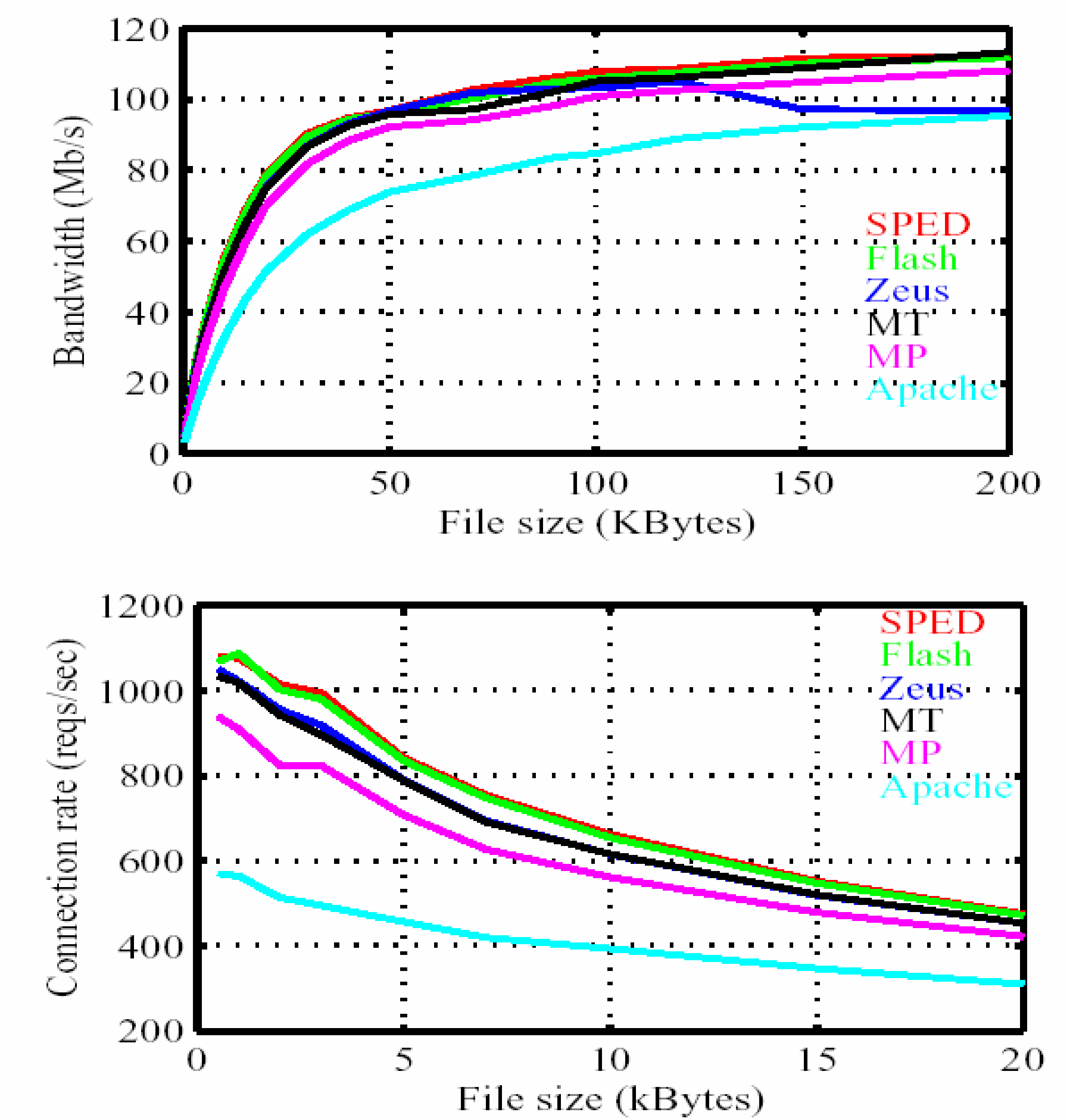


- Main ideas:
  - main server process handles memory resident requests (like SPED)
  - helper processes handler concurrent blocking operations (like MT/MP) (smaller footprint compared to entire process)
  - overhead: check before file read if memory resident, IPC communication with helper threads
- An interesting new issue (w.r.t. the web server in the project) is caching. Cached pages can be handled well by the main thread (e.g., boss thread in an MT design)
  - from s-m require support for mincore/mlock to lock pages and verify that they are in memory;

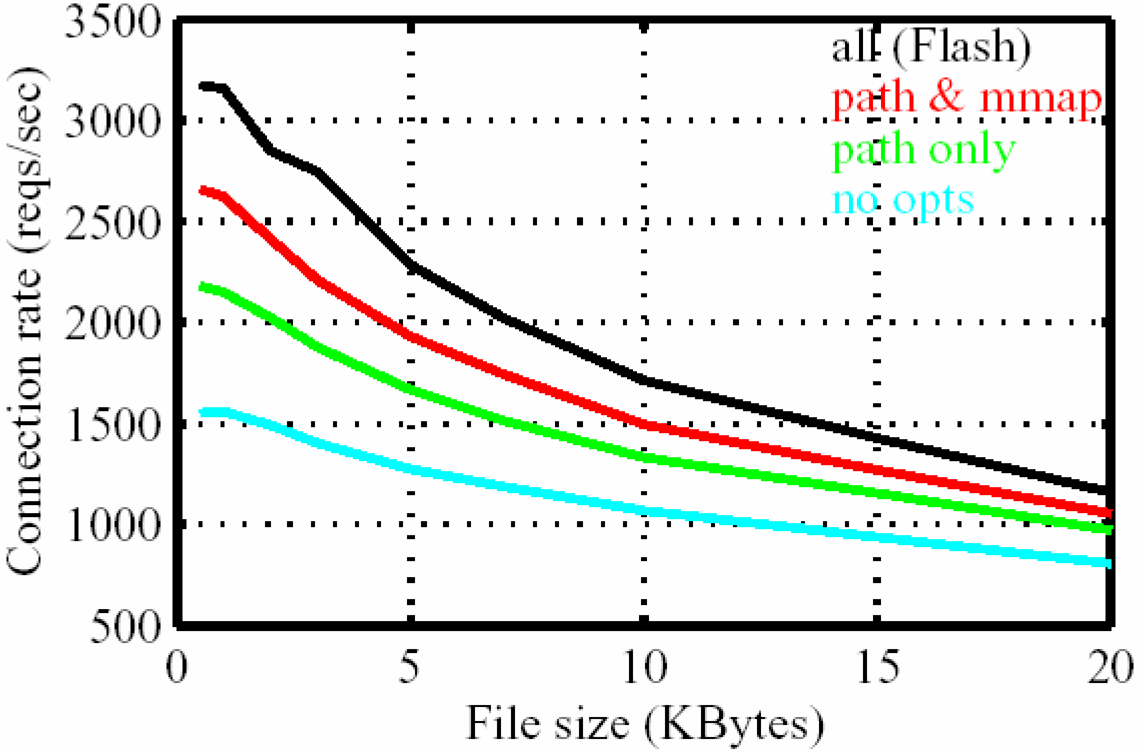
performance benefits

- smaller memory requirements than a MT/MP server
- better information sharing – no synchronization needed for access to shared data.
- application-level caching (headers, path resolutions…)
  - LRU policy for unmapping chunks
- gather writes (e.g., header + payload)
  - issue with byte alignment

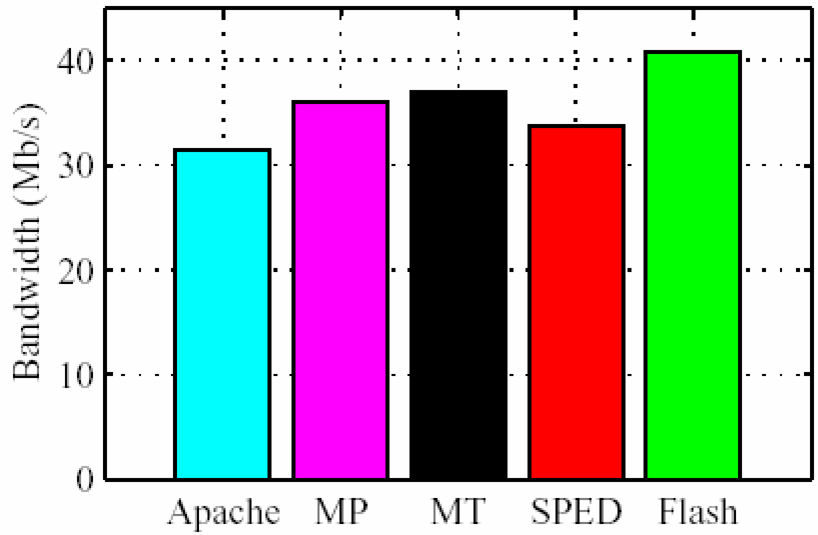

Start
Accept Conn
Read Request
Find File
Pathname Trans. Cache
Helper
Send Header
Response Header Cache
Read File Send Data
Mapped File Cache
Helper
End



- best case number (same file from cache)

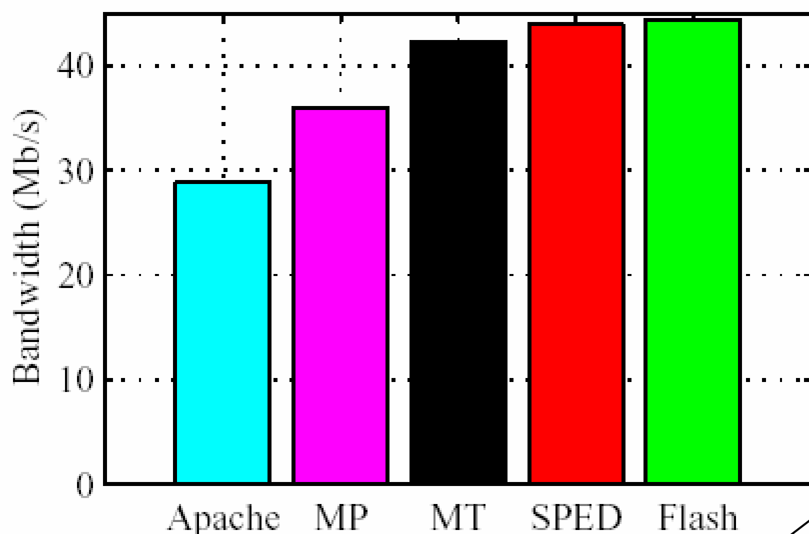


all (Flash)
path & mmap
path only
no opts
Connection rate (reqs/sec)
3500
3000
2500
2000
1500
1000
500
0
5
10
15
20
File size (KBytes)



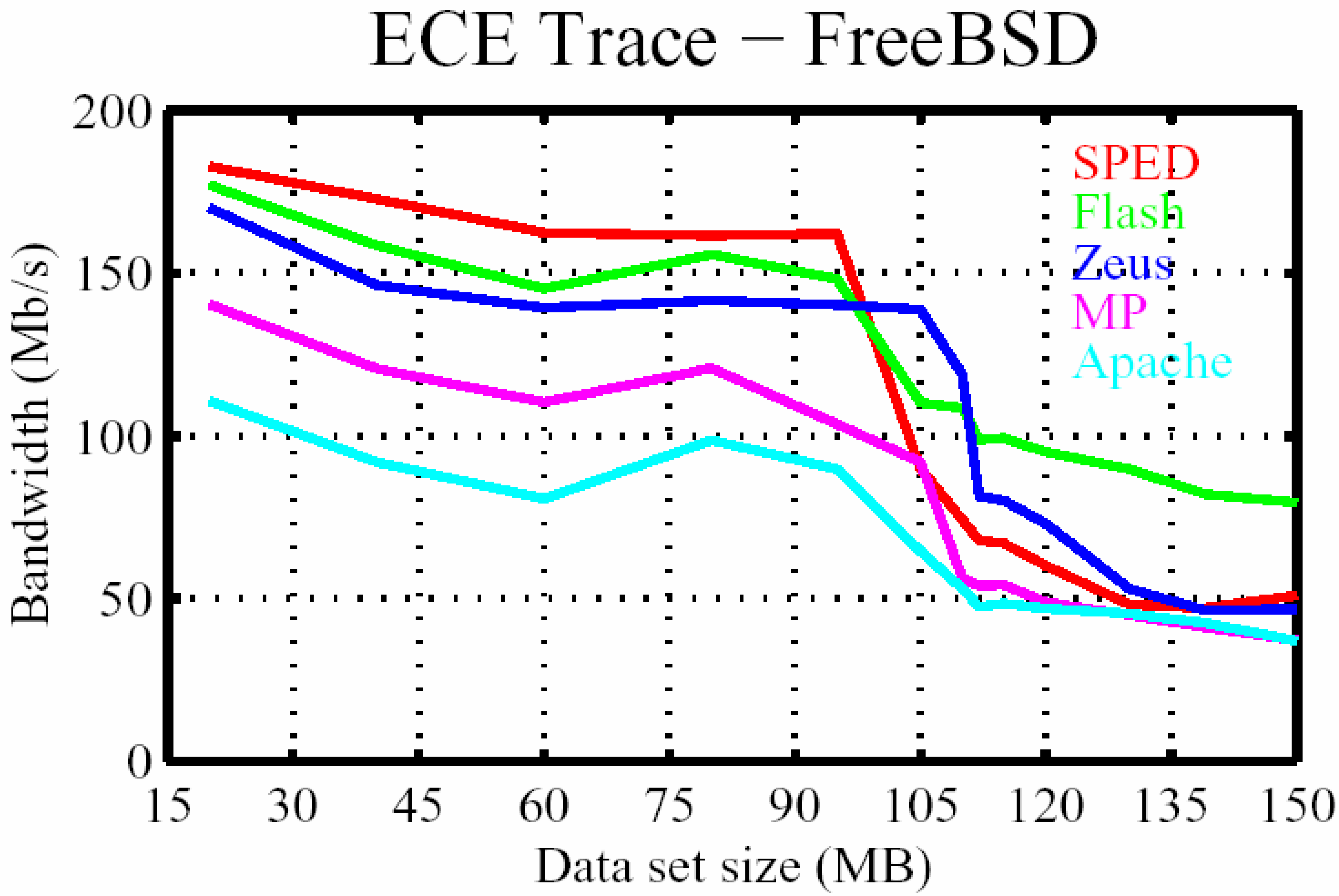
CS trace

IO bound for this trace

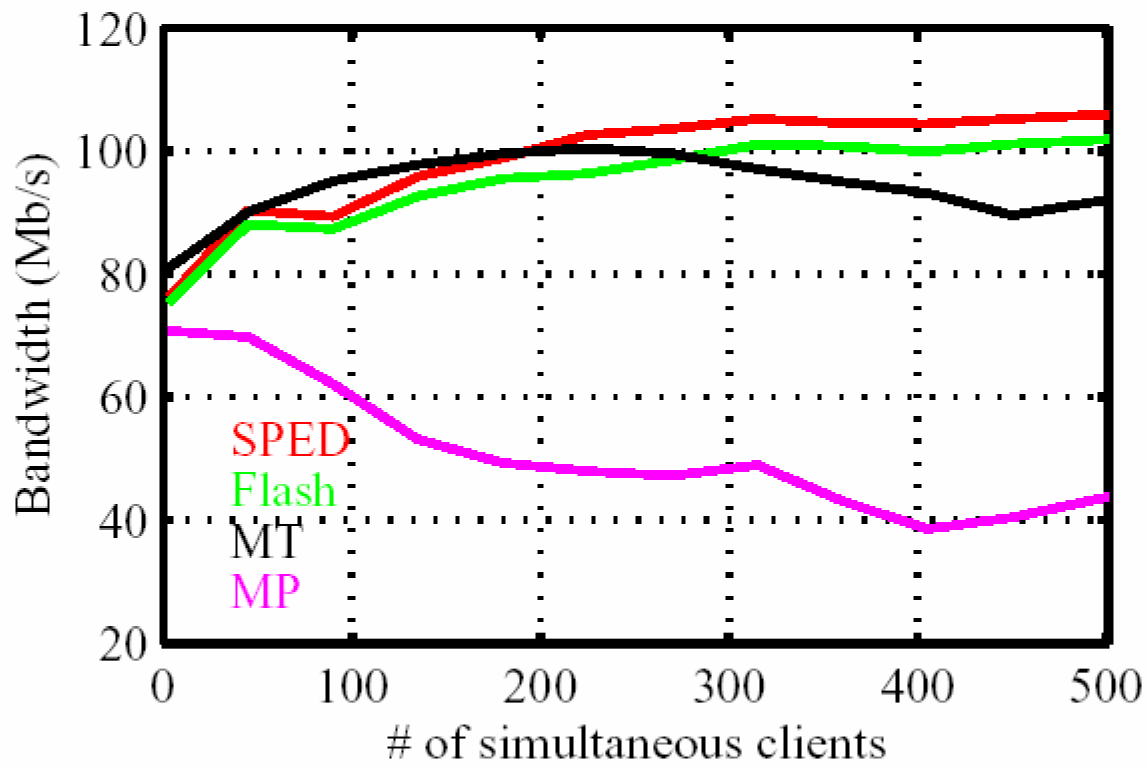


Owlnet trace

smaller trace fits in Mm



ECE Trace − FreeBSD
200
150
100
50
0
Bandwidth (Mb/s)
15
30
45
60
75
90
105
120
135
150
Data set size (MB)
SPED
Flash
Zeus
MP
Apache



- in WAN setting #concurrent clients becomes important

summary of performance eval

- when data in cache SPED >> AMPED Flash (unnecessary tests for memory residence)
- both >> MT/MP – context switching overhead
- with disk-bound workloads, AMPED Flash
  - >> SPED (SPED blocks)
  - >> MT/MP (more memory efficient, and less context switching – there are multiple threads/processes, only when need to do disk I/O.

how do multiple processes perform listen?

- listen on separate ports on different N/W I/Fs
- listen on one socket – all block on accept, TCP/IP stack is such that it will wake them all up, and give the connection only to one.
  - others will spin the kernel, then go to sleep
  - better to actually serialize calls to accept! (e.g. reported performance measurement – serialization reduces req/sec for < 3%, but lack of it adds 100ms latency on each requst)
- select(..) on multiple socket

```
for (;;) {
     for (;;) {
        fd_set accept_fds;
        FD_ZERO (&accept_fds);
        for (i = first_socket; i <= last_socket; ++i) {
           FD_SET (i, &accept_fds); }
        rc = select (last_socket+1, &accept_fds,
                     NULL, NULL, NULL);
        if (rc < 1) continue;
        new_connection = -1;
        for (i = first_socket; i <= last_socket; ++i) {
           if (FD_ISSET (i, &accept_fds)) {
             new_connection = accept (i, NULL, NULL);
             if (new_connection != -1) break;
           }
        }
        if (new_connection != -1) break;
     }
        process the new_connection;
}
```

```
for (;;) {

      accept_mutex_on();

      for (;;) {
         fd_set accept_fds;

        … same as before …

        if (new_connection != -1) break;
      }
      accept_mutex_off();

      process the new_connection;
}
```

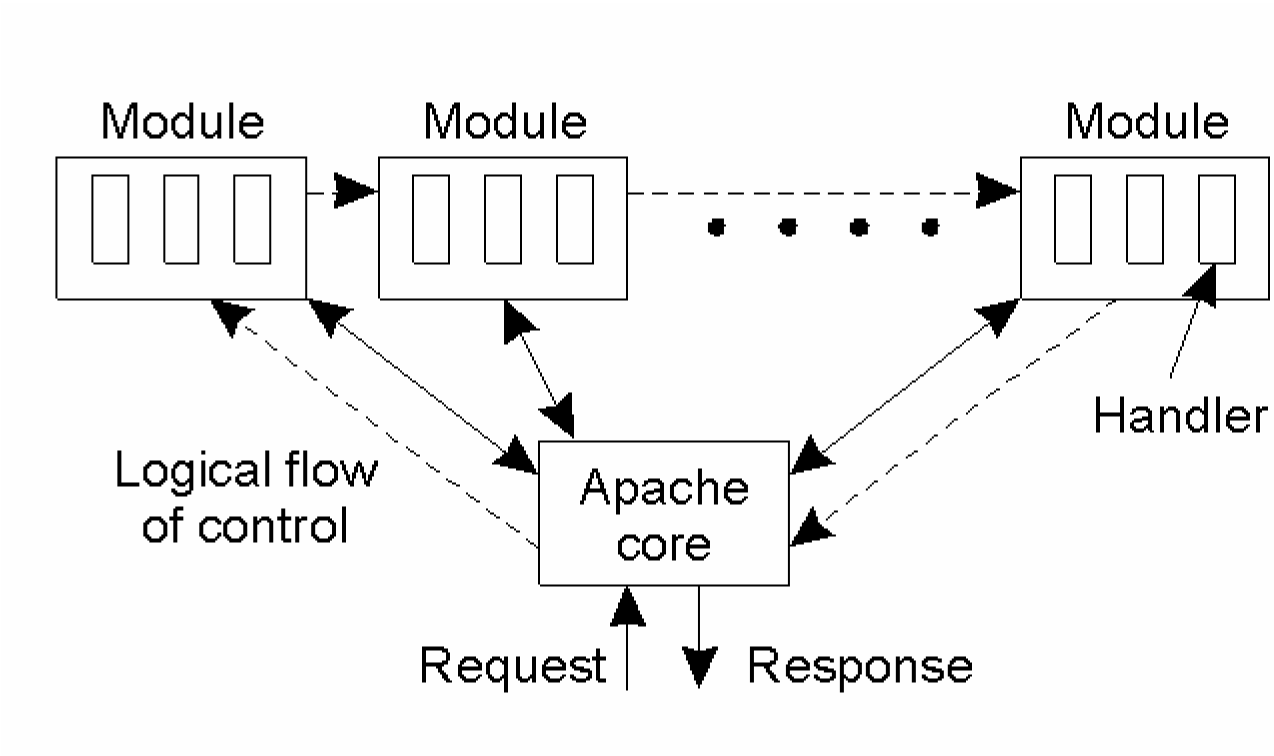
- mutual exclustion is implemented with mechanism available on the underlying system – semaphores, POSIX mutexes, `flock()`/`fcntl()` system call…

Apache model

- similar – server core, and modules…
- a requests is processed in phases, each phase is initialized by the core
- a request record associated with request, each module operates of it in specific order
- model is more implicit invocation, as opposed to event-driven, core invokes handler in a module, receives response, than invokes next handler…
- different requests handled by different child processes
  - (actually, new Apache releases are MT)
  - different techniques can be configure to vary number of processes/threads
    - basically number varies between low/high watermark

Servers

- General organization of the Apache Web server.

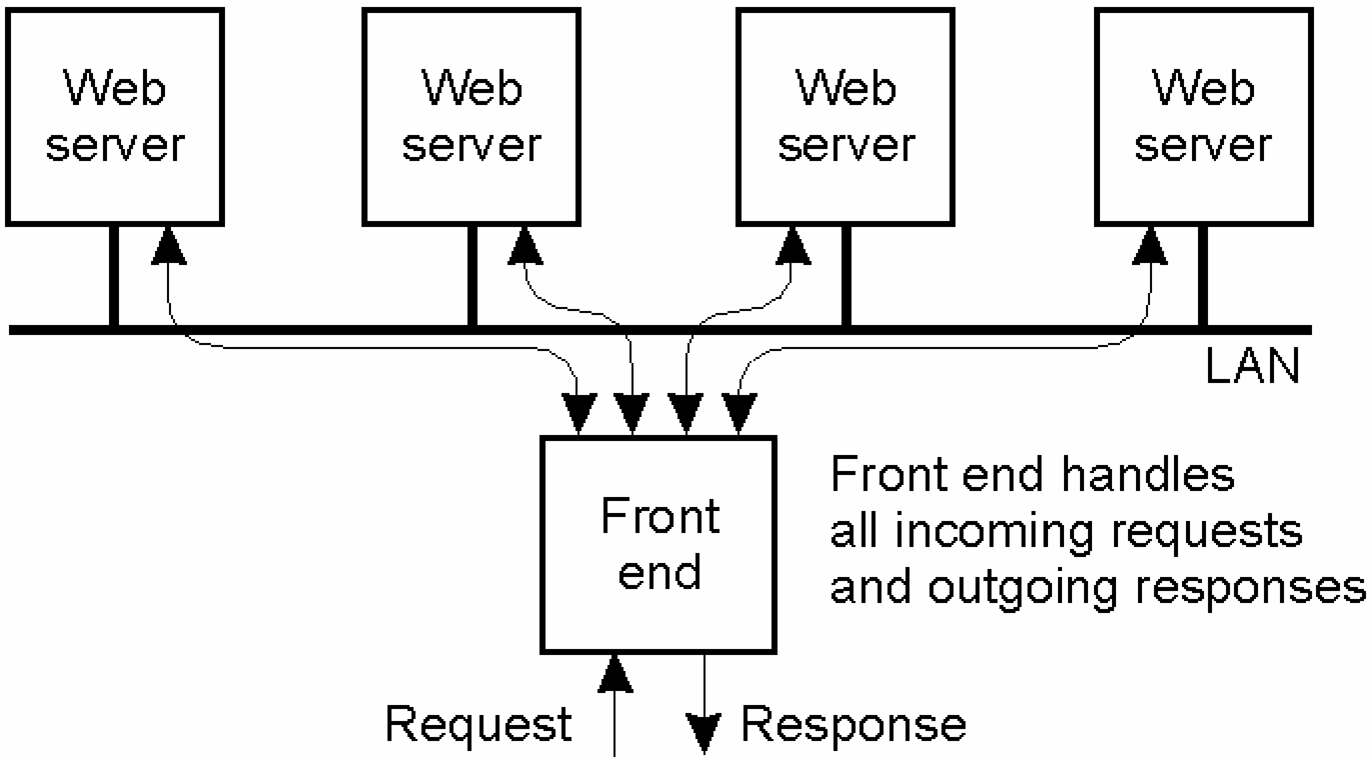


Cluster Servers

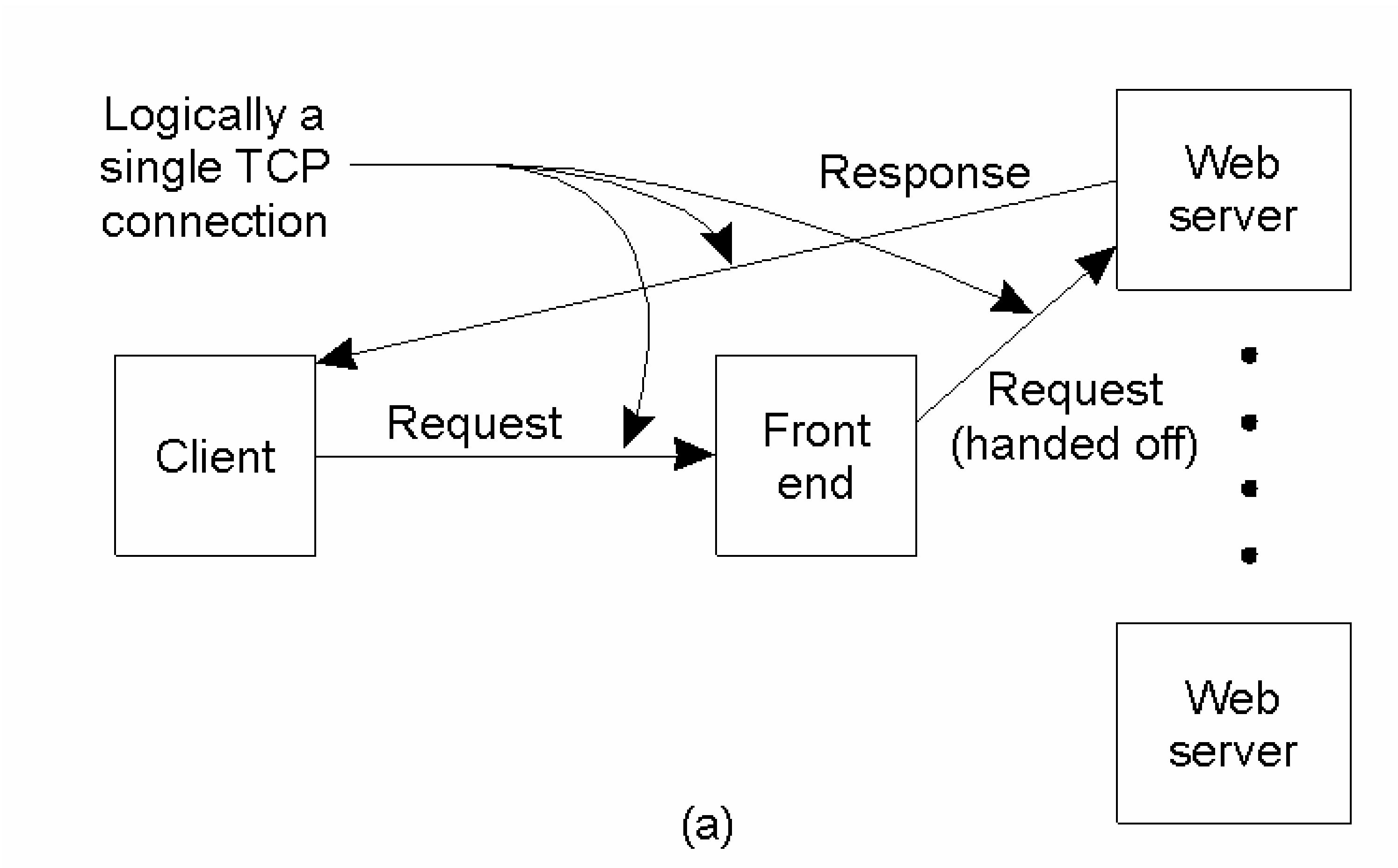
- Front end accepts requests and forwards request to one of the cluster nodes
  - transport-layer switch, typically based on some load balancing policy
  - application-layer front-end – for content-based request distribution (request type, specific values, some ‘caching’ probability…)
- TCP handoff – cluster node returns directly to client, client unaware that it might be talking to a different endpoint.

Server Clusters

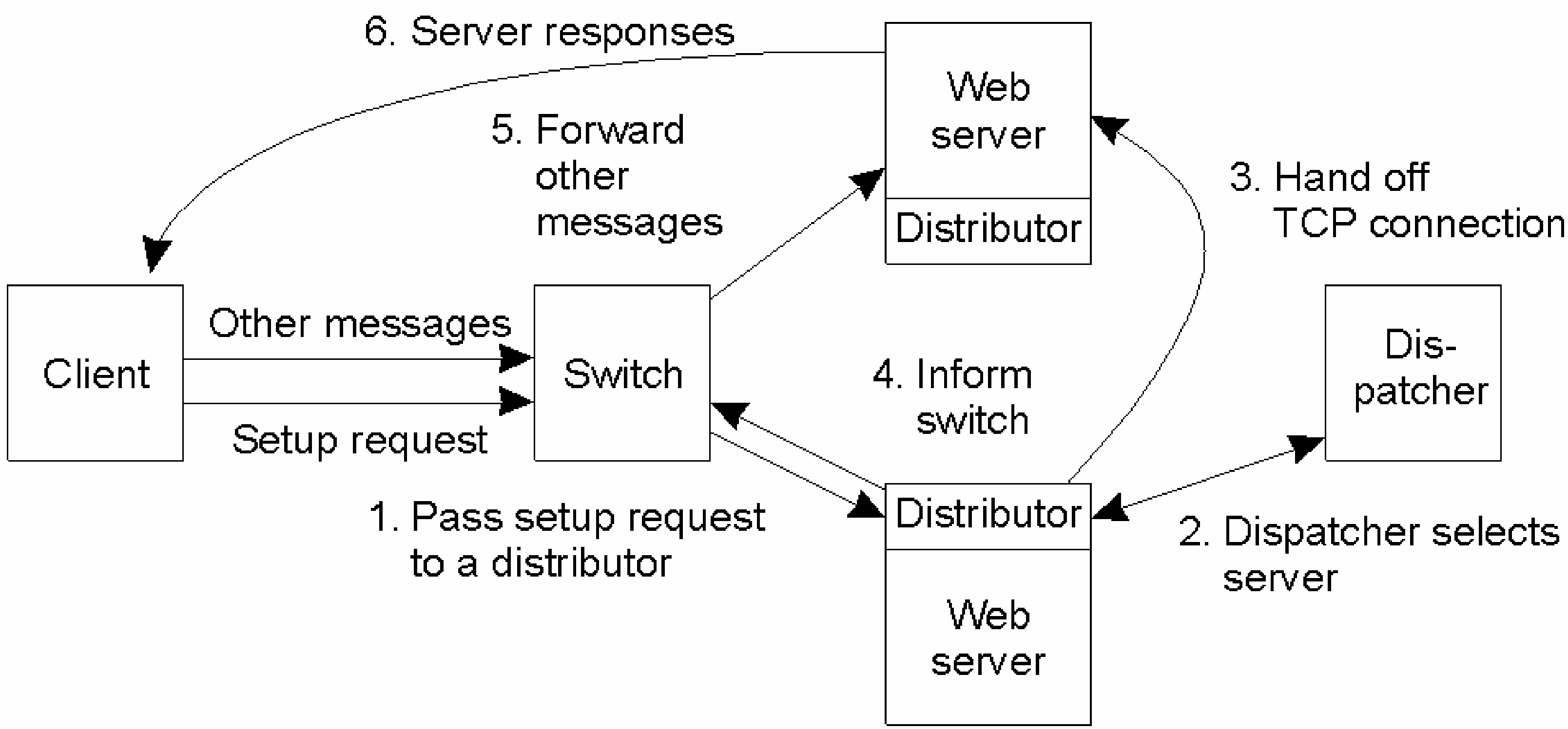
- The principle of using a cluster of workstations to implement a Web service.



TCP handoff


Logically a single TCP connection
Response
Web server
Client
Request
Front end
Request (handed off)
Web server
(a)

a scalable content-aware cluster of Web servers transport switch + 'smart' cluster node selection + handoff



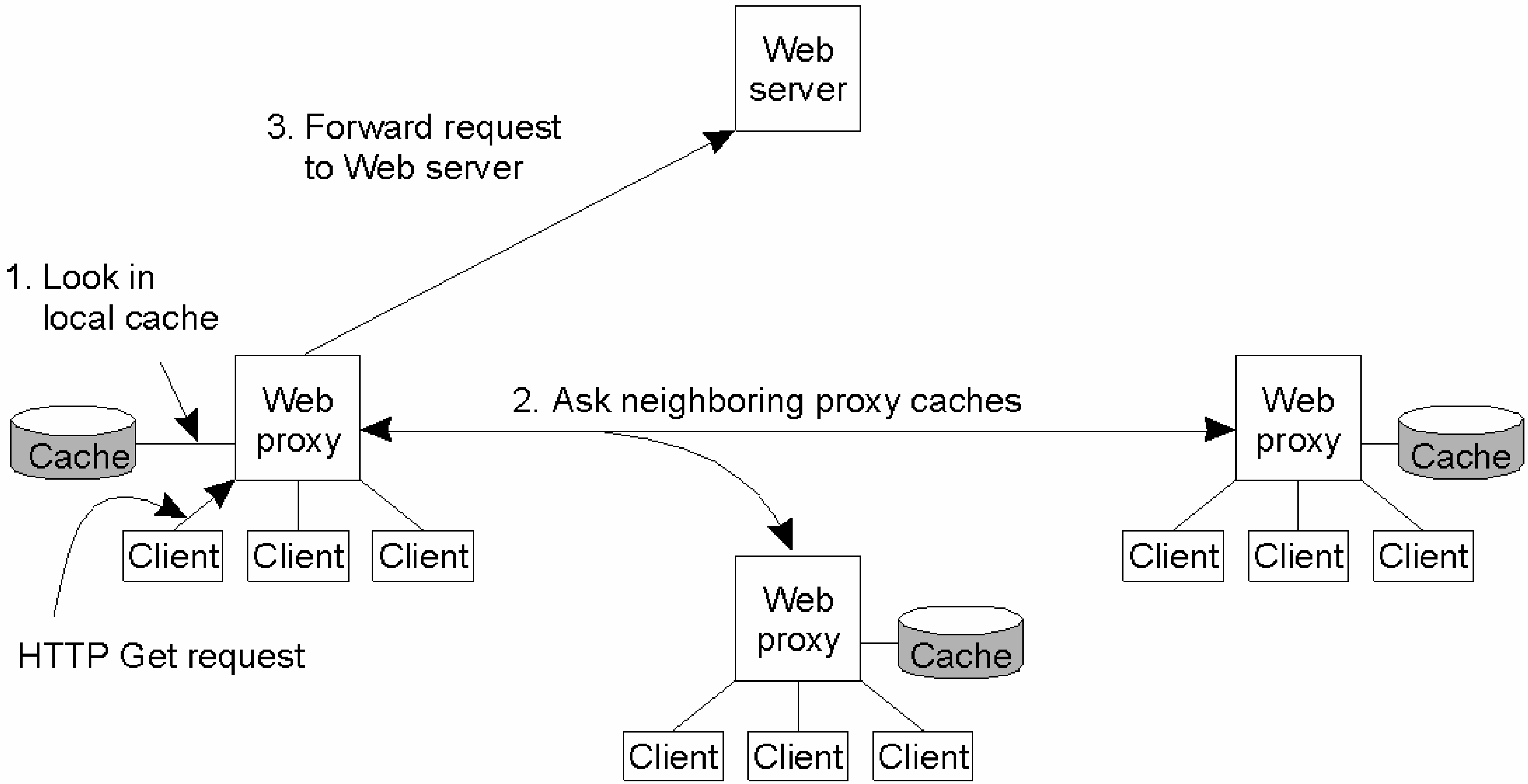
(b)

Other issues

- Proxy caching
- Mirrored servers – client explicitly knows to talk to a different mirror server
- Content Delivery Networks (CDNs)

Web Proxy Caching

- The principle of cooperative caching



Server Replication

- The principle working of the Akami CDN.