Distributed concurrency

Runtimes for concurrency and distribution

Tullio Vardanega, tullio.vardanega@unipd.it

Academic year 2021/2022

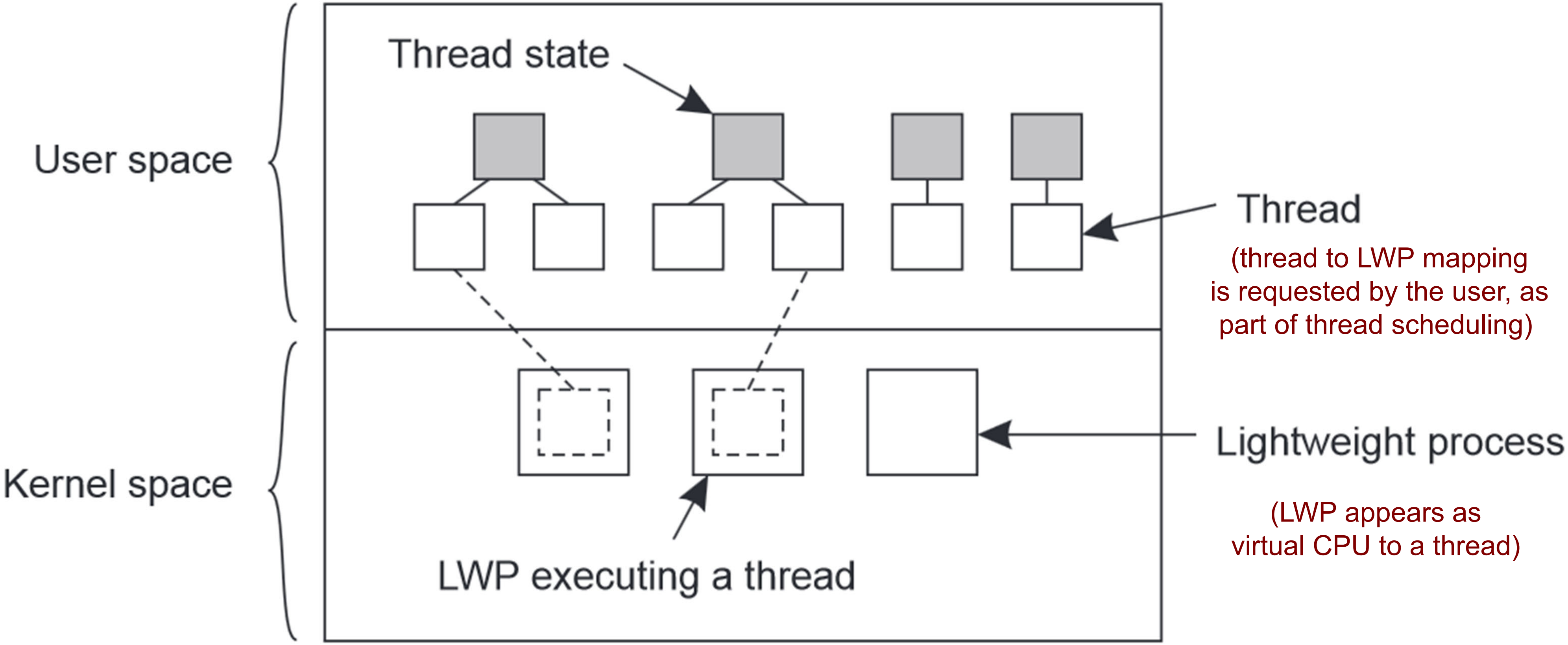
Appreciating the cost of abstractions – 1

- **Processor context**
  - The processor registers
    - A few tens (16, 32, 48) in the general case
- **Thread context**
  - The processor context
  - The stack, their share of heap, the thread descriptor
  - Creating and switching threads begins to be costly
- **Process context**
  - The context of all threads
  - The virtual-memory page frames assigned to the process, the corresponding descriptors
  - Creating and switching processes is very costly

Appreciating the cost of abstractions – 2

- Thread-level context switch may be fairly nimble as long as it does *not* involve the OS
  - The OS gets involved on blocking IO calls or when external events (interrupts, signals, …) have to be delivered to a thread
  - When that happens, the whole process may be blocked
- Threads need *not* be OS entities
- Several user-space to kernel-space mappings are possible
  - **Many:1** (old GNU)
    multiple user threads to one kernel thread → no thread-level parallelism
  - **1:1** (old Win, old Linux)
    one user thread to one kernel thread → the OS does all the scheduling
  - **Many:Many** (Win NT, Solaris Unix)
    multiple user threads *dynamically* to multiple *lightweight processes* (**LWP**), which can be statically allocated → LWPs may run in parallel

Appreciating the cost of abstractions – 3

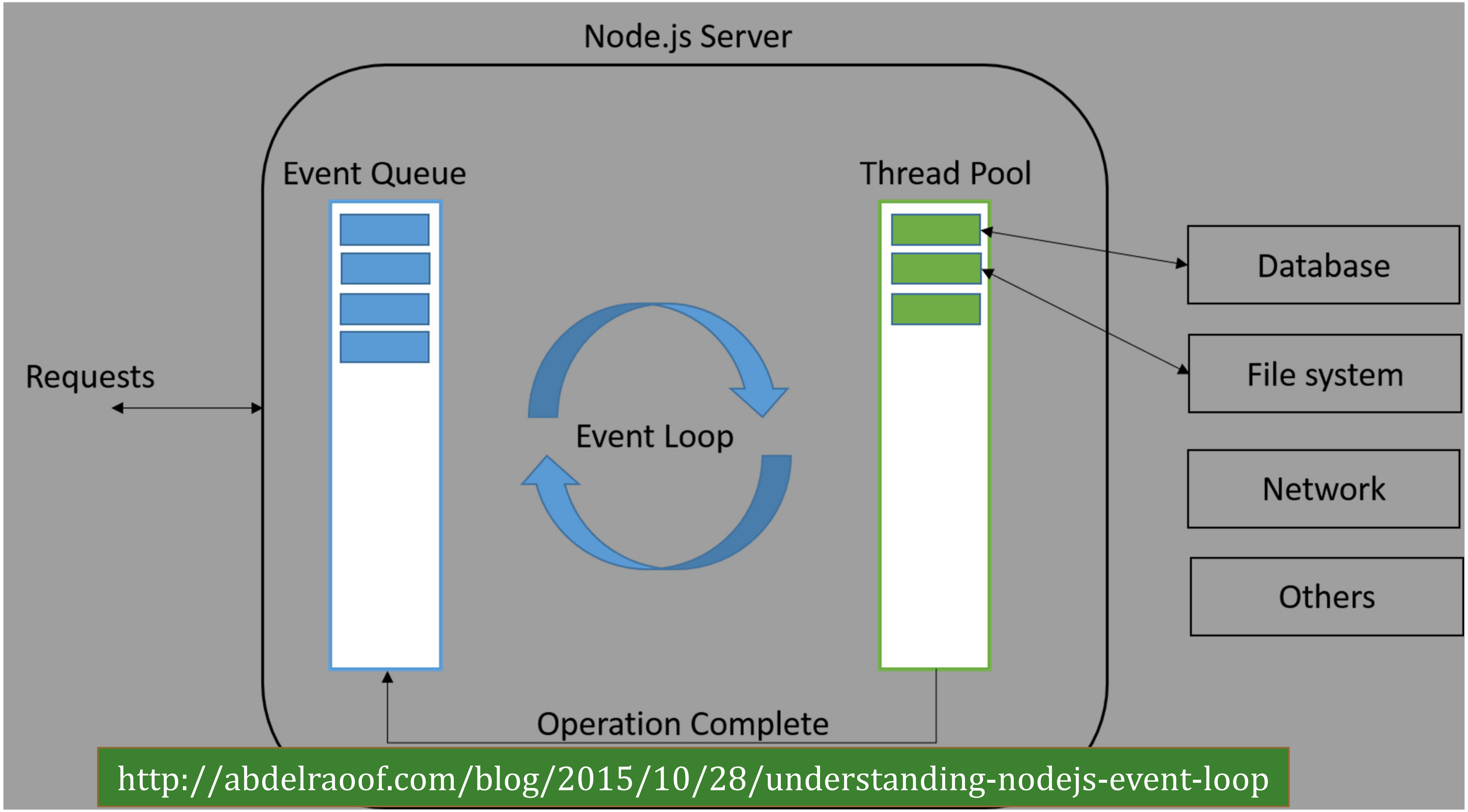


C.G. Ritson, A.T. Sampson, F.R.M. Barnes, *Multicore scheduling for lightweight communicating processes*, Science of Computer Programming 77(6), June 2012, DOI: 10.1016/j.scico.2011.04.006

Appreciating the cost of abstractions – 4

- Server realized as kernel process may underestimate the cost and the limits of dynamic thread creation
- **Example**: the **Apache** Web Server used to deploy one thread per connection
  - The service capacity of a WS process is upper bounded by the maximum number of threads that it can embed …
  - The cost-benefit ratio of a 1:1 thread-to-connection mapping depends on the data volume being transported
    - Used to be large data volumes for few connections in Web 1.0
    - Became tiny data volumes for very many connections in Web 2.0
- Using threads for IO-bound computations is wasteful
  - **Node.js** understands this notion very well …

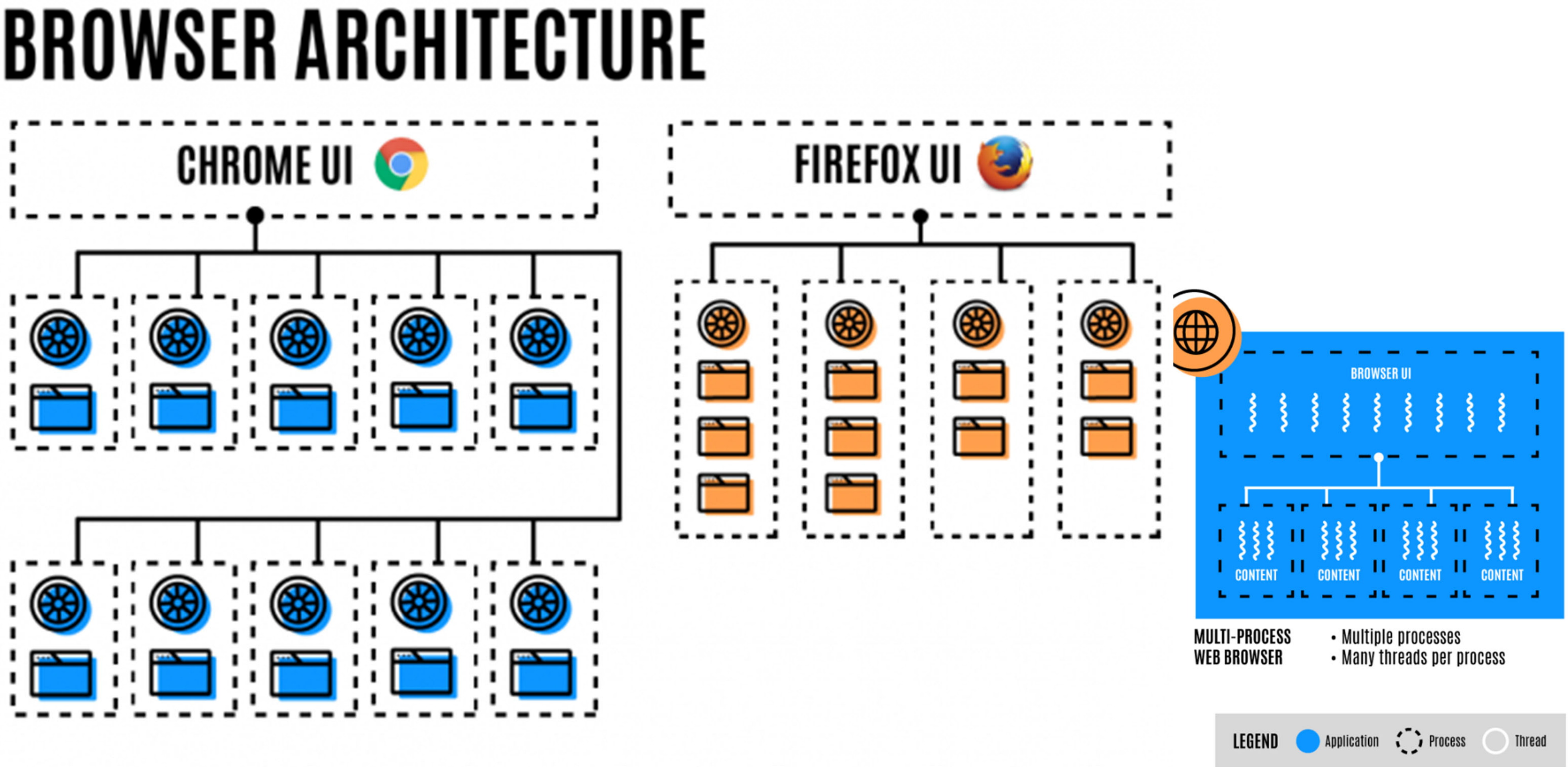
Appreciating the cost of abstractions – 5



Client-side concurrency

- Helps mitigate network delays
  - Very evidently needed in web browsers
    - Starting a TCP connection is a blocking and slow operation
    - Requesting data and rendering them are pipelined
  - **AJAX** (*Asynchronous JavaScript And XML*) came to be precisely to enable asynchronous page updates
- Google Chrome was the first browser to go multithreaded (2008), Firefox since v54 (2017)
  - Recent Chrome used one kernel process per tab
  - Recent Firefox used one kernel process for the first few (4) tabs, then one thread for any further tab

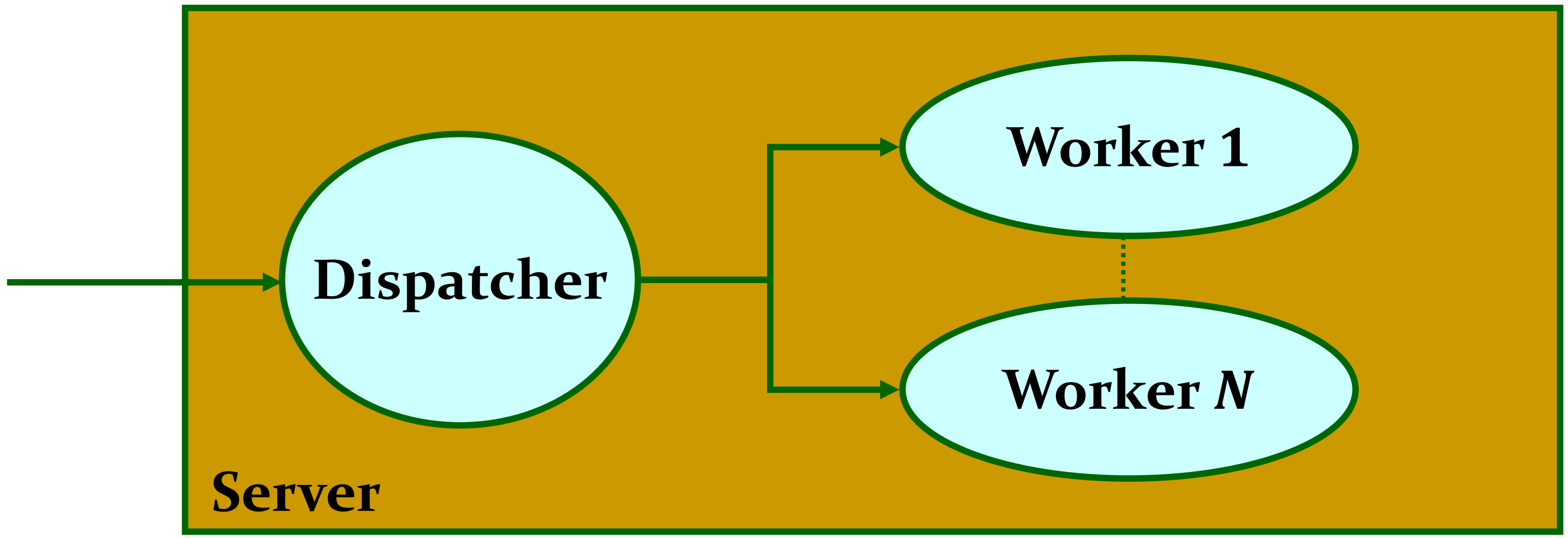
Chrome vs Firefox



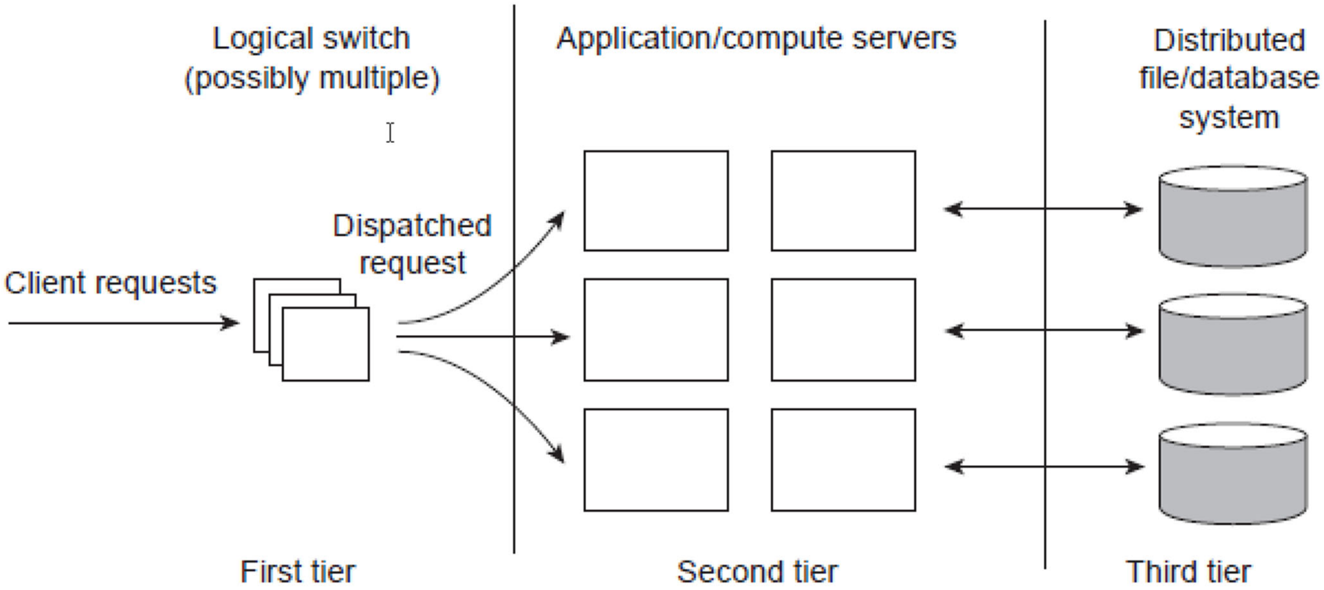
www.extremetech.com/internet/250930-firefox-54-finally-supports-multithreading-claims-higher-ram-efficiency-chrome

Server-side concurrency – 1

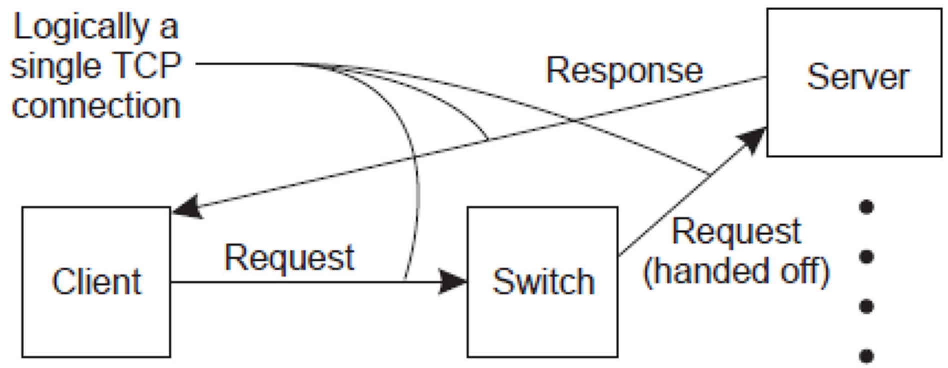
- For higher throughput and better modularity
- The obvious base architecture is two-level



Server-side concurrency – 2

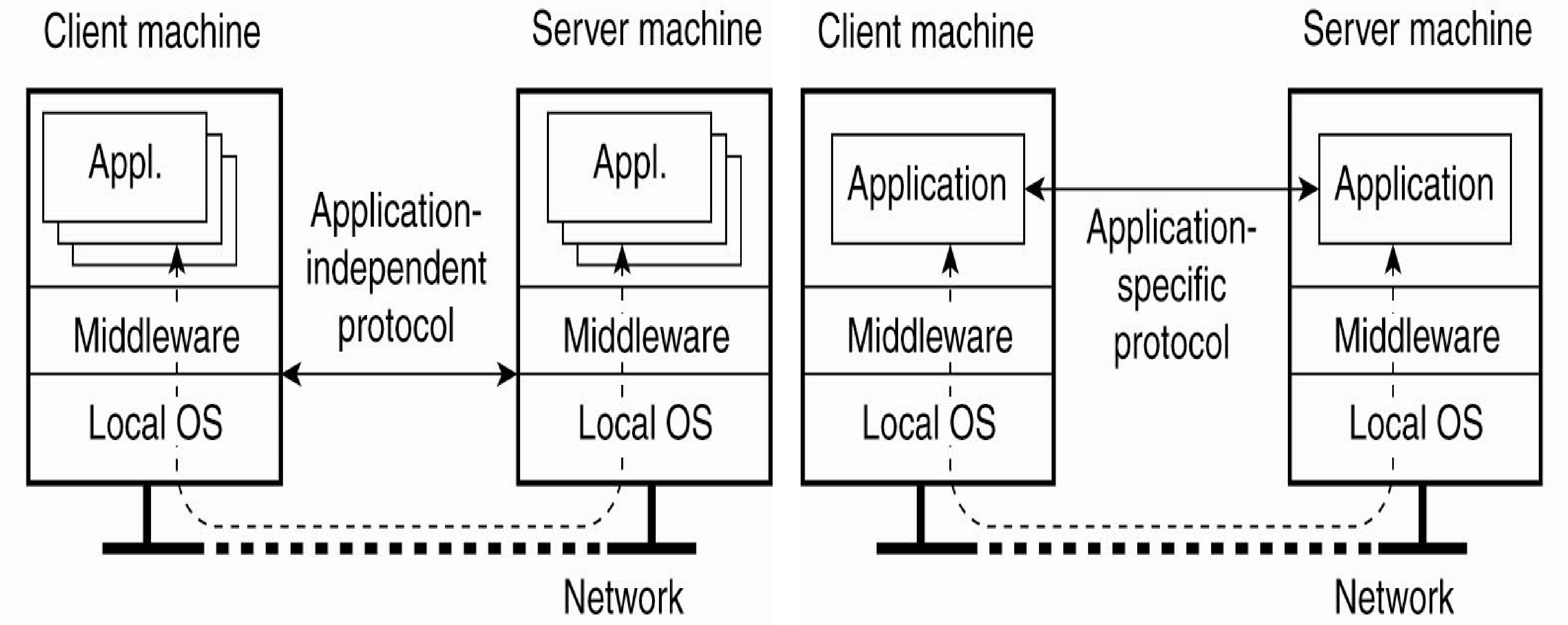


TCP hand-off relieves 1st level receiver



Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Client-side features – 1



Thin-client architecture

Fat (Thick)-client architecture

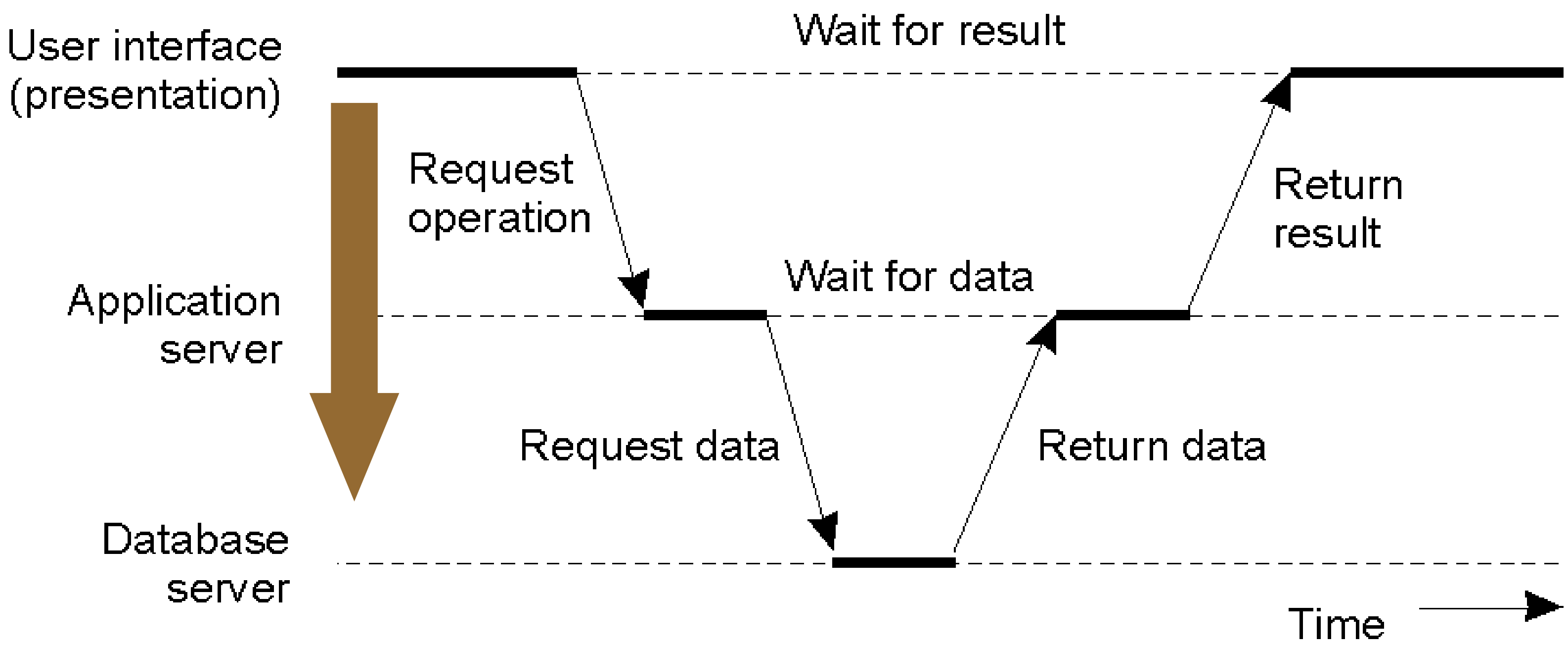
Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Client-side features – 2

- **Thin clients** are fed by *application-neutral* communications
  - Server side decides all; client side is unable to mitigate server lapses
  - The choice of **X11** (X Window System, xorg)
- **Fat clients** are fed by *application-specific* communications
  - The client side may have things to do without the server dictating them
  - More responsive for the user, lighter for the server
- How can we classify **single-page web apps**?

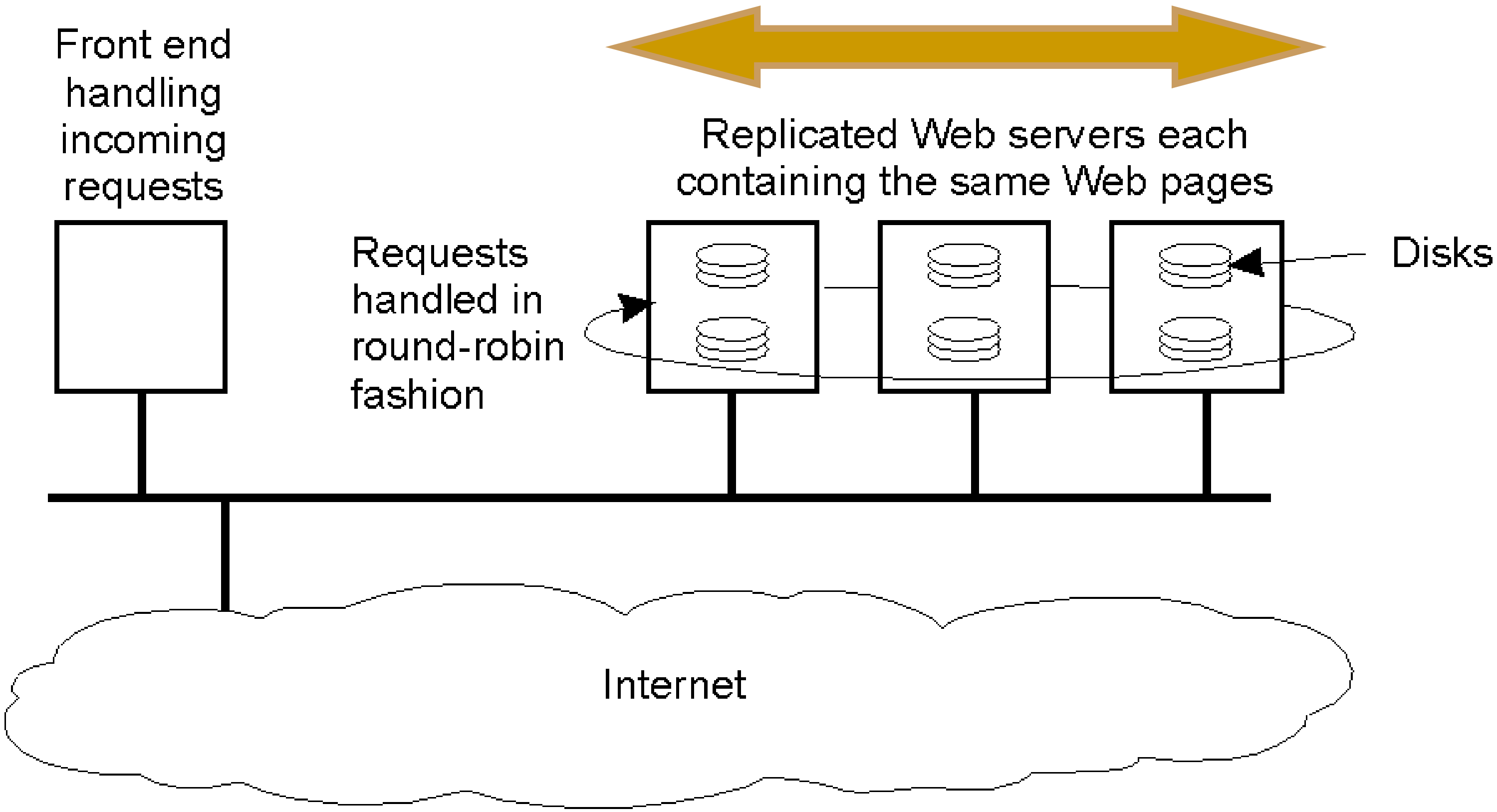
Server-side organization – 1

- **Vertical distribution**
  - Service provision is split in *synchronous* stages
  - New inbound requests are held until completion of current service
  - Full server replication required to improve throughput



Server-side organization – 2

- **Horizontal distribution**
  - Very fit for idempotent services …



Server-side organization: microservices

In Pursuit of Architectural Agility: Experimenting with Microservices

Published in: 2018 IEEE International Conference on Services Computing (SCC)

Date of Conference: 2-7 July 2018

INSPEC Accession Number: 18076512

Date Added to IEEE *Xplore*: 06 September 2018

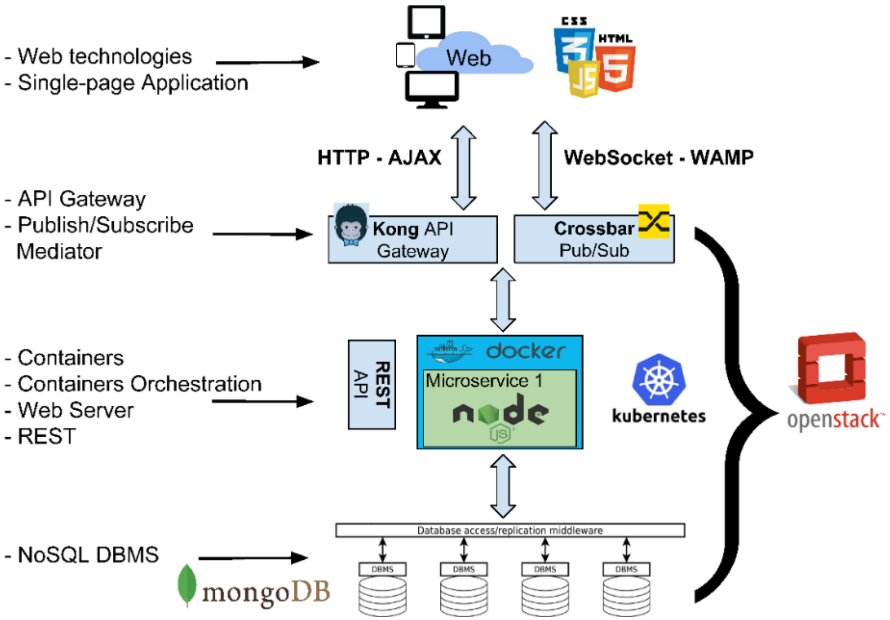
DOI: 10.1109/SCC.2018.00022

▸ **ISBN Information:**

Publisher: IEEE

Electronic ISSN: 2474-2473

Conference Location: San Francisco, CA, USA



II. The microservices approach: a short recap

The term "microservices" designates an architectural style that yields a single application from the coordination of a suite of unitary services [5]. Such services expose an Application Program Interface (API) *outside* of their codebase (a central trait of their specific composition style), which the user invokes using *asynchronous* (crucial to loose coupling) *web-based* service requests (key to reachability).

A microservice is understood as a small self-contained application that has a single responsibility, a lightweight stack, and can be deployed, scaled and tested independently

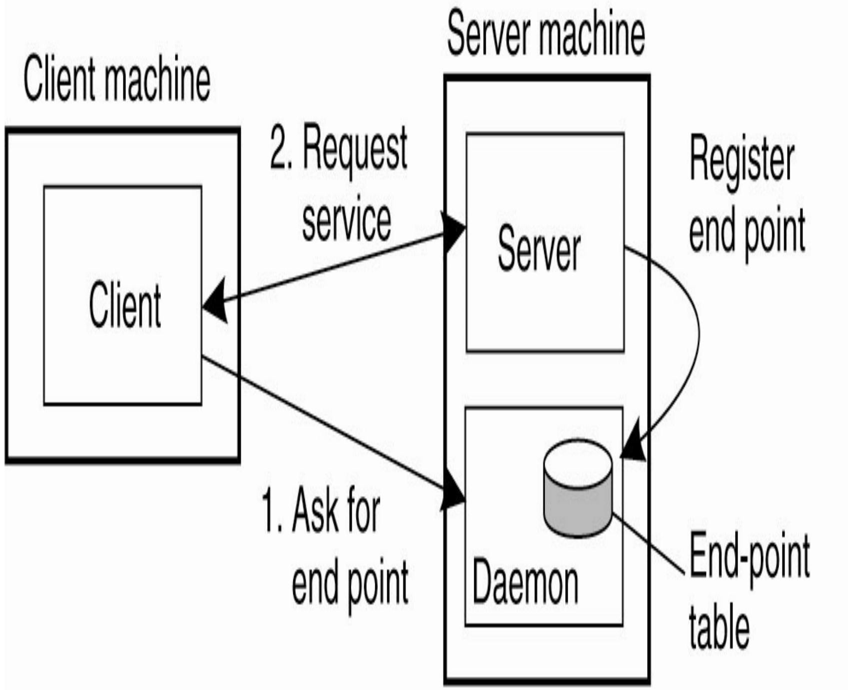
Microservices in practice

- Key architectural concepts of a Microservice architecture (WSO2)
  - https://wso2.com/whitepapers/microservices-in-practice-key-architectural-concepts-of-an-msa/
- A reference architecture at WSO2
  - https://github.com/wso2/reference-architecture/blob/master/api-driven-microservice-architecture.md
- An interesting toy example
  - https://github.com/FudanSELab/train-ticket

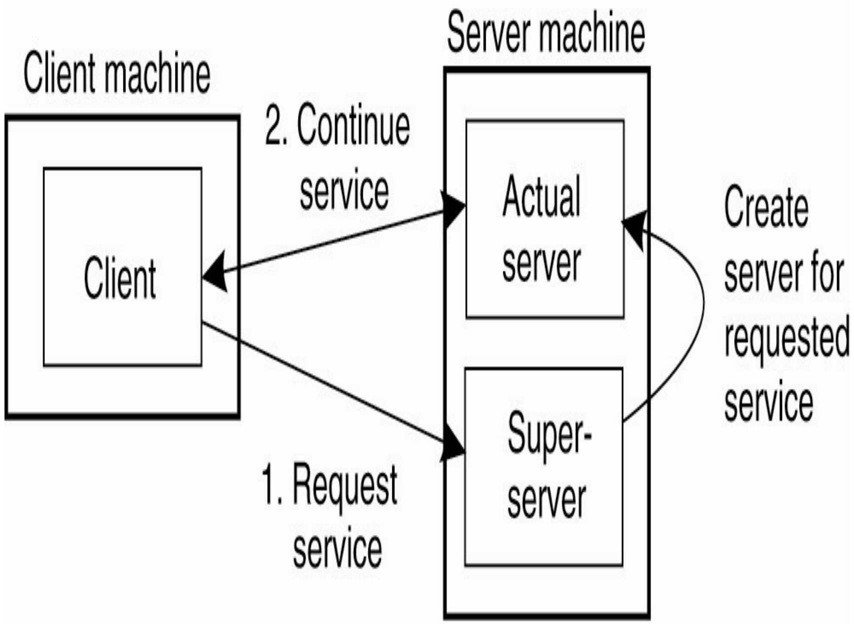
Server localization – 1

- Server identified by ***endpoint*** at its host node
  - {IP address : port, object reference}
  - A dedicated process must listen on the corresponding port and then dispatch the call to the associated server object
- Per-node port assignment is a challenge
  - The **IANA** (*Internet Assigned Numbers Authority*) statically assigns some to base common servers
  - All others have to resort to dynamic assignment
    - A *daemon* listens on an assigned port and assigns them dynamically as needed to the servers it handles
    - A *super-server* (e.g, `inetd` in Linux) listens on a set of “server ports” and then dynamically hands off to newly-created server

Server localization – 2



Daemon-based solution



Super-server solution

Tanenbaum & Van Steen, *Distributed Systems: Principles and Paradigms*, 2e, (c) 2007 Prentice-Hall, Inc.

Server state – 1

- **Stateful** servers warrant state consistency to clients
  - All clients sense the same write history
- Transactional DBs are the most prominent exemplar of that paradigm
  - $\boldsymbol{begin}\ (Op_1, Op_2, \ldots, Op_n)\ \boldsymbol{commit}$
  - **Atomicity**: state change is all-or-nothing
  - **Consistency**: the server state is always the product of ordered transactions $(Op_1, Op_2, \ldots, Op_n)$
  - **Isolation**: concurrent transactions do not overlap
  - **Durability**: the effect of successful transactions persists
- **Transactions centralize: they cannot scale**

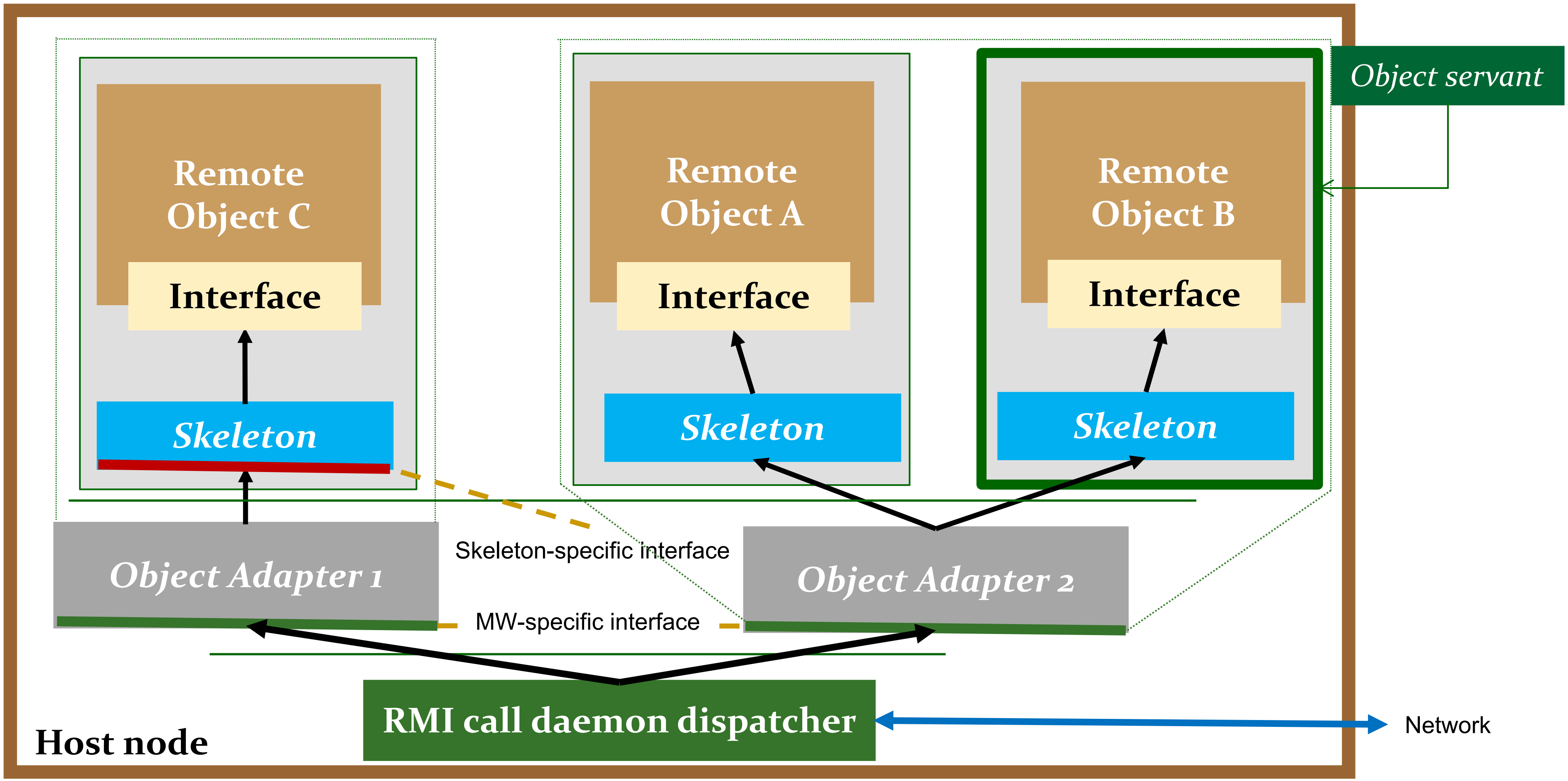
Server state – 2

- **Stateless** servers do *not* inform clients of any server-side state change
- They also do *not* retain client-side service state across connections
  - This is what caused *cookies* to come to use
- **NFS** was the most prominent exemplar of it
  - Client operates locally on *virtual inode* with *write-through* local cache (not coherent across clients)
  - Server handles each individual request without memory of client-side state
  - Server-side state may change outside of clients’ knowledge
- **Statelessness is crucial to elastic scalability!**

RMI: object servant – 1

- Remote object (server) lives in a scope managed by an "*object servant*" that has authority over it
  - Servant holds server state and supports a range of **activation policies** for it at run time, which determine server's life cycle
    - Create / destroy object (server) reference part of server's endpoint
    - Provide / revoke computational resources for the server
- The activation policies of multiple servants on the same host node can be factored in an **object adapter** (OA)
  - OA pattern uses *interface delegation*
  - Single per-node receiver of inbound RMI calls to multiple resident remote objects
  - Single per-node registry of object servants
  - Single MW-specific interface on one end, multiple object-specific interface on the other

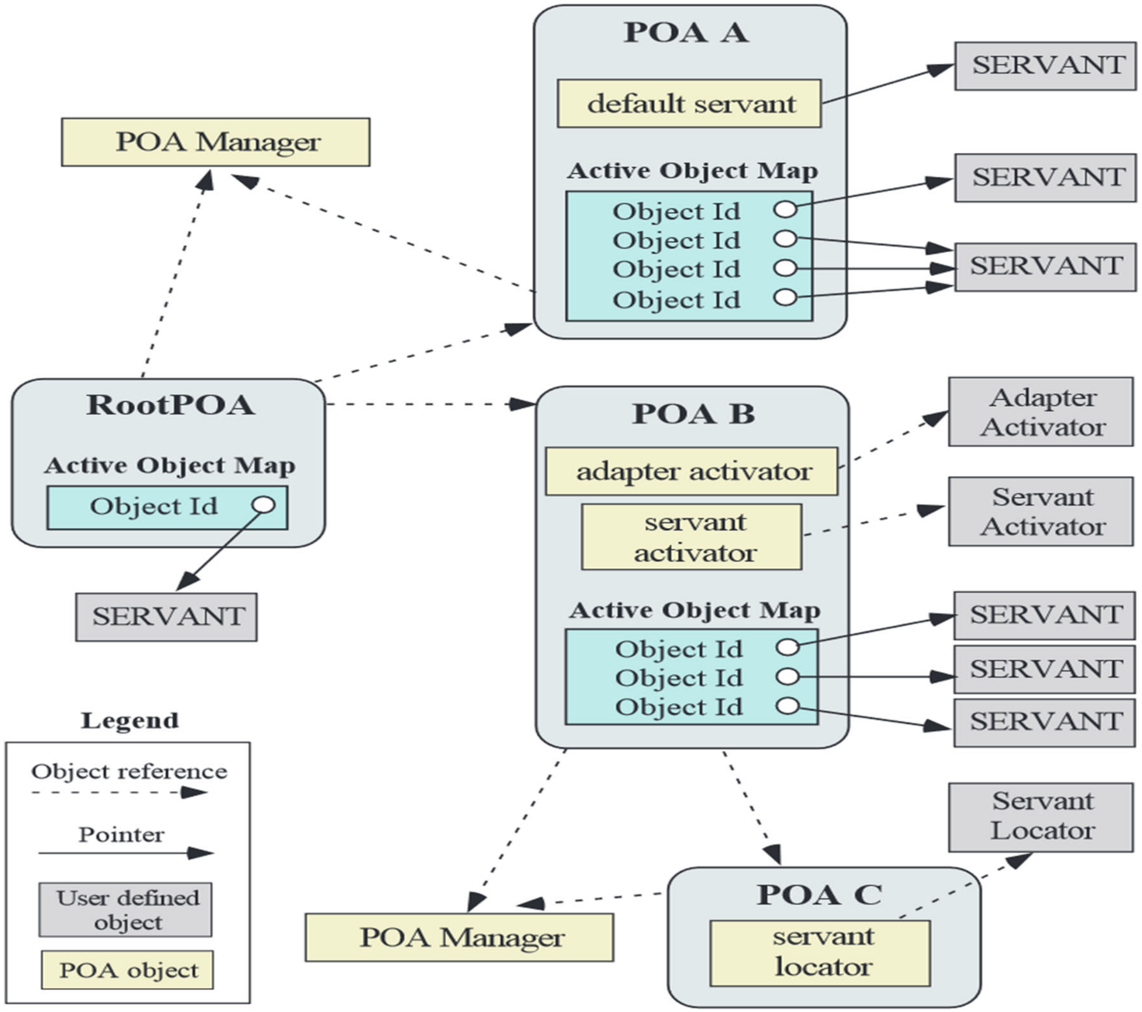
RMI: object servant – 2



RMI: object servant – 3

- The OA must expose a *standard* interface to the part of the program's middleware that listens to the service endpoint
  - Totally *independent* of the target RMI interface
- The skeleton must expose a *standard* interface to the OA that has to deliver incoming calls to it
  - Generic, not specific to the target RMI interface

CORBA's Portable Object Adapter



Pyarali & Schmidt, An Overview of the CORBA Portable Object Adapter