03: Client-Server Architectures

Distributed Software Architectures

Stefan Huber <shuber.lba@fh-salzburg.ac.at>

April 4, 2019

Section 1

Client-Server Architectures

Concurrency models for servers

A server has to handle multiple clients concurrently:

- Connections may be short-term: connect, request, response, close.
- Connections may be long-term or permanent, e.g., a gaming server.
- HTTP connections used to be short-term, but became long-term.

There are two main models for concurrency:

Model	**I/O model**
Fork on request	Synchronous I/O, blocking
Event-driven	Asynchronous I/O, nonblocking

Fork on request

Concurrency can be provided on two levels:

- Processes
- Threads

General scheme:

- Parent process or main thread accepts connections.
- Child process or worker thread handles client connections.

The term *fork* refers to the POSIX function `fork()` to fork[1] a process. But we also account for threads here.

[1] Like cell mitosis in biology.

Fork on request: C-style forking TCP server

```
1  import socket, os
2
3  if __name__ == "__main__":
4      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
5      s.bind(("", 1200))
6      s.listen()
7
8      while True:
9          conn, addr = s.accept()
10         print("Got conncetion from", addr)
11         newpid = os.fork()
12
13         # The child
14         if newpid == 0:
15             s.close()
16             conn.send("hello there".encode())
17             exit(0)
18         # The parent
19         else:
20             print("Forked child", newpid)
21             conn.close()
```

Fork on request: Python-style TCP server

The module `socketserver` provides

- `TCPServer`
- `UDPServer`

By default single-threaded, synchronous. But there are MixIns:

- `ThreadingMixIn`
- `ForkingMixIn`

```
1  import socketserver
2
3  class MyHandler(socketserver.BaseRequestHandler):
4      def handle(self):
5          self.data = self.request.recv(1024).decode()
6          self.request.sendall("got: {}".format(self.data).encode())
7
8  class MyTcpServer(socketserver.ThreadingMixIn, socketserver.TCPServer):
9      pass
10
11 if __name__ == "__main__":
12     s = MyTcpServer(("localhost", 1202), MyHandler)
13     s.serve_forever()
```

Fork on request

Forking processes is costly:

- Setting up process memory, heap, stack, and kernel data structures.

A process context switch comes at costs:

- Switches between kernel and user space.
- Saving processor registers
- Switching to a different virtual memory mapping and flushing the TLB[2].
- A thread context switch is cheaper, but still comes at costs.

Level	**Provider**	**Register dump**	**VMM switch**
Process	Kernel	Yes	Yes
Thread	Kernel or user space	Yes	No

Observation

If we want to handle many thousand connections per second, we cannot afford the costs for starting threads or processes or switching context between them.

[2] Translation lookaside buffer

Event-driven architecture

What we start with:

- We want concurrency.
- We cannot handle each connection in a dedicated thread or process.

Hence,

- we stick with a single thread only and
- we cannot use blocking, synchronous I/O calls.

[3] The send buffer must not be full.

[4] The receive buffer must not be empty.

[5] Files, sockets, et cetera.

Event-driven architecture

What we start with:

- We want concurrency.
- We cannot handle each connection in a dedicated thread or process.

Hence,

- we stick with a single thread only and
- we cannot use blocking, synchronous I/O calls.

Non-blocking, asynchronous I/O calls:

- `send()` and `recv()` immediately return.
- But we need to know whether we can send[3] or receive[4].
- The `select()` call monitors a set of file descriptors[5] and returns when at least one becomes ready for read or write.

Summary

We use non-blocking I/O routines and I/O multiplexing based on `select()` mechanism to handle multiple concurrently with a single thread.

3 The send buffer must not be full.

4 The receive buffer must not be empty.

5 Files, sockets, et cetera.

Event-driven architecture: I/O multiplexing

The Python module `selectors` provides an abstraction to the OS primitives[6]:

- `DefaultSelector` watches a set of file-like objects.
- `select()` gives a list of ready file objects.
- `register(fileobj, events, data=None)` makes the selector to watch for the given events (read ready, write ready) on the given file object. The data supplied here is returned by `select()`.
- `unregister(fileobj)` unregisters the given file object.
- `close()` closes the entire selector and frees resources.

[6] There are actually many.

Event-driven architecture: I/O multiplexing

```
import selectors, socket

if __name__ == "__main__":
    sel = selectors.DefaultSelector()
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(("", 1300))
    s.listen()
    s.setblocking(False)

    def read(sock, mask):
        data = sock.recv(1024)
        if data: # Bad: We actually do not know whether we can write!
            sock.send("** ".encode() + data)
        else:
            print(sock.getpeername(), "disconnected")
            sel.unregister(sock)
            sock.close()

    def accept(sock, mask):
        conn, addr = sock.accept()
        print(addr, "connected")
        conn.setblocking(False)
        sel.register(conn, selectors.EVENT_READ, read)

    sel.register(s, selectors.EVENT_READ, accept)
    while True:
        for key, mask in sel.select():
            key.data(key.fileobj, mask)
```

Event-driven architecture

```
# The event loop
while True:
  # Get list of pending events (file become ready-ready or write-ready)
  for key, mask in sel.select():
      # mask can be
      #   selectors.EVENT_READ or
      #   selectors.EVENT_WRITE or
      #   selectors.EVENT_READ | selectors.EVENT_WRITE
      # key.fileobj is the file object for which the event occured
      # key.data is the data that has been passed to register()
      key.data(key.fileobj, mask)
```

We interpret this loop as the event loop:

- We register file objects (e.g., sockets) as event sources.
- An event can be `selectors.EVENT_READ` or `selectors.EVENT_WRITE`.
- `sel.register(sock, selectors.EVENT_READ | selectors.EVENT_WRITE, data)` makes `sel` to listen for both kind of events on `sock`.

Event-driven architecture

The code below is insufficient:

- The socket may not be ready for `send()`.
- We then get an `BlockingIOError`.
- We have to register with `selectors.EVENT_WRITE` enabled and when the socket is ready to write fetch data from the buffer.

```
def read(sock, mask):
    data = sock.recv(1024)
    if data:
        # Bad: We actually do not know whether we can write!
        # We actually need to store the response in a buffer
        # and wait until we receive the EVENT_WRITE event for
        # sock!
        sock.send("** ".encode() + data)
    else:
        print(sock.getpeername(), "disconnected")
        sel.unregister(sock)
        sock.close()
```

Event-driven architecture

Summary:

- No thread or process context switch between handlers of different connections.
- In a single event loop iteration we may accept resp. handle a plethora of new resp. existing connections.
- Concurrency logic is moved from the system (operating system, system libraries) to the application. This increases code complexity compared to the simple fork-on-request model.

Preforking

Forking processes is costly:

- Hence, do not fork on request but before (pre) requests arrive.
- That is, a master process initially forks a pool of worker processes that handle upcoming requests.
- The number of worker processes is typically fixed, but could be adapted to load.

Synchronous I/O:

- Preforking and blocking I/O does not deliver (full) concurrency. We can only handle as many clients as workers in the pool.
- Can be adequate for some application, maybe for a file server where too many disk I/O operations cannot be handled anyway.

Asynchronous I/O:

- An event-driven architecture with non-blocking I/O does not use preforking for concurrency.
- However, with multiple workers we can increase CPU utilization.

NGINX architecture

NGINX:

- An event-driven webserver with focus on high-performance.
- Used by 37.7% of top 1M websites, 49.7% of the top 100k and 57% of the top 10k websites in 2016-21-19.[7]
- March 2019: F5 Networks acquired Nginx company for 670M USD.
- Article on NGINX architecture: `https://www.aosabook.org/en/nginx.html` (2012-05-01)

High concurrency:

- Popular website serves 100k to 1M users simultaneously.
- Causes of concurrency needs:
  - Decade ago: Slow clients.
  - Now: Persistent connections for live update for apps and browsers (tweets, news, feeds). Modern browsers opening 4–6 simultaneous connections.

[7] `https://w3techs.com/technologies/cross/web_server/ranking`, current data show a further shift to NGINX.

NGINX architecture

Lifetime of a connection:

- A slow client (10 kB/s) requests a 100 kB response. Transmission takes 10 s.
- 100 new connections per second cause 1000 simultaneous connections.
- If a server needs 1 MB main memory[8] per per client this sums up to 1 GB in total.
  - Maybe, to transmit the *same* 100 kB to all 1000 clients.
- Persistent connections of modern web make the situation much worse.

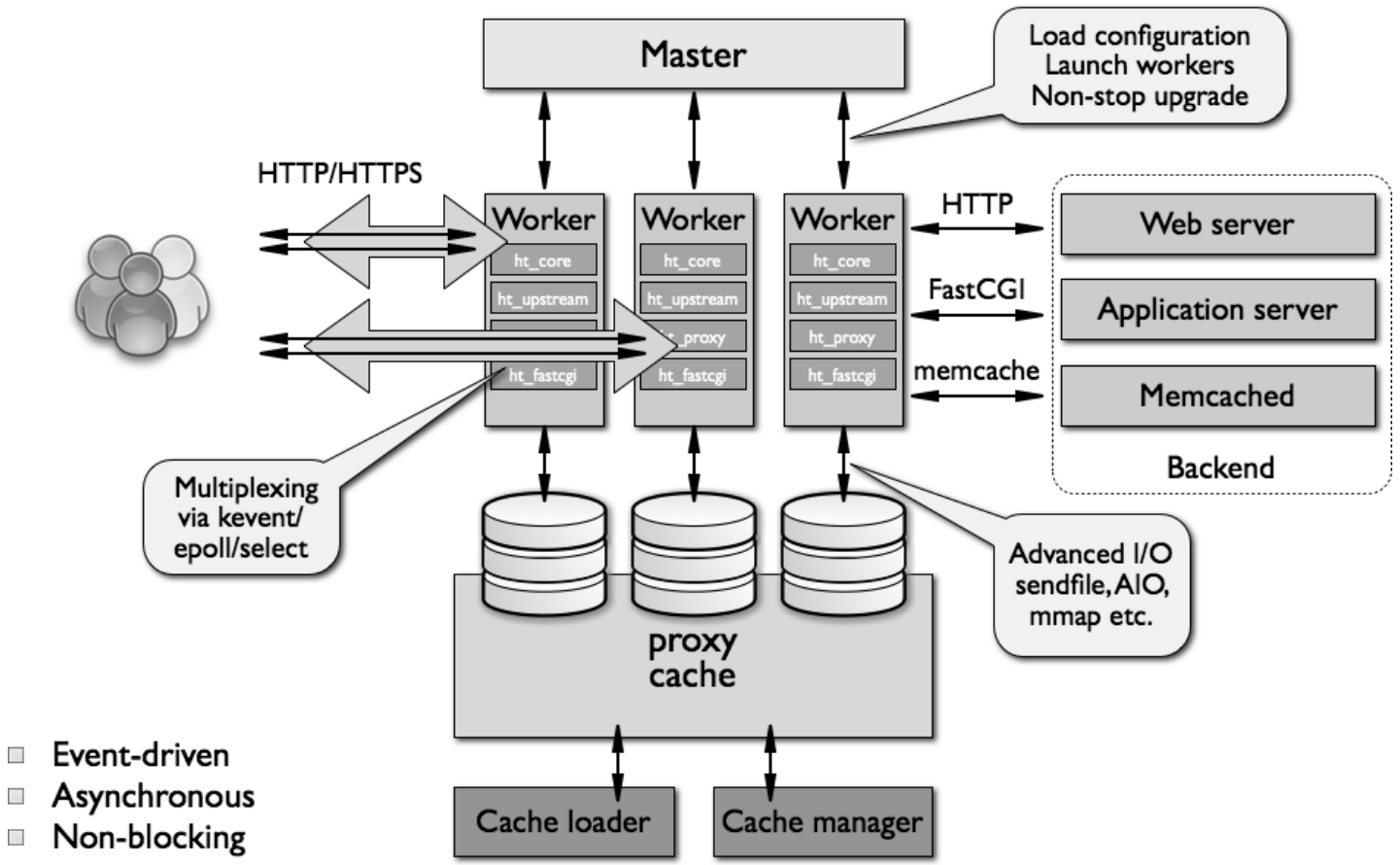
Kegel's C10K manifest (1999):

- Webservers shall be able to handle 10k simultaneous connections.
- 2004: NGINX initially released to address the C10K problem.
- 2012: WhatsApp on 24 core machine, Erlang on FreeBSD: 2M connections[9]

[8] For instance, kernel data structures for 1000 child processes.

[9] `http://www.erlang-factory.com/upload/presentations/558/efsf2012-whatsapp-scaling.pdf`

NGINX architecture



- Master forks workers. Workers drop privileges and run the async I/O event loop.
- 2.5 MB memory footprint per 10k (inactive, keep-alive) HTTP connection.
- Often used as load balancer and frontend for backend servers or reverse proxy.
- IPC is primary shared-memory mechanism.

Centralized system organization

A *centralized organization* has the following characteristics:

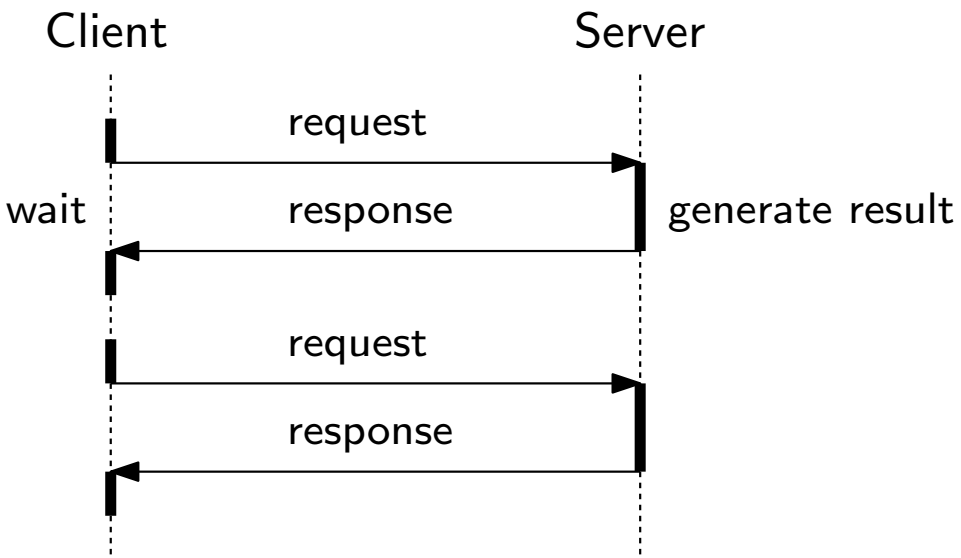
- A central service is provided by servers.
- Clients are using the service provided by servers.

A simple 2-tiered server-client architecture with a request-response communication is one example of an centralized organization.

Communication patterns

Request-response:

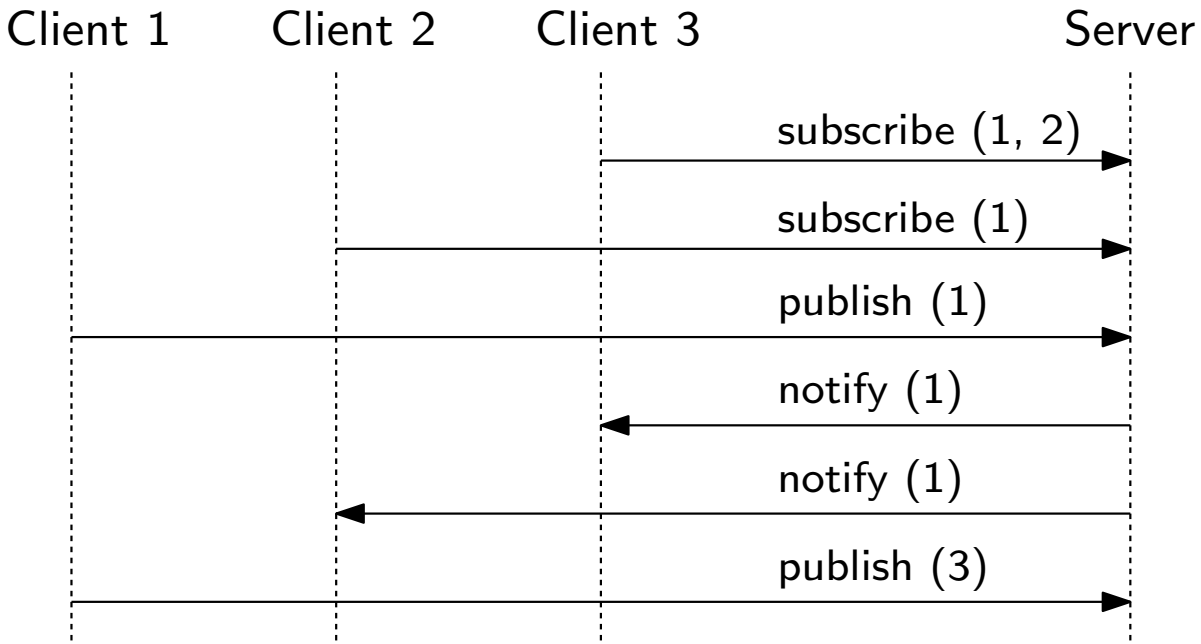
- Client sends request to server and server answers with the response.
- Communication is often connection-less or a connection with a short lifetime.
- Setting up connection is costly, hence multiple request-response sequences may be done over one connection.
- Connection-less communication needs to deal with message losses. Simpler if operations are *idempotent* because they can be simply retransmitted "Tell me amount on bank account" versus "Transfer this amount to this account".



Communication patterns

Publish-subscribe:

- Clients subscribe and/or publish messages of certain categories.
- Server receives messages and forwards them on to clients.
- Clients connect permanently or listen to connection-less broadcasts.
- Modern web (e.g., social media updates) show characteristics of publish-subscribe.



Two-tiered architecture

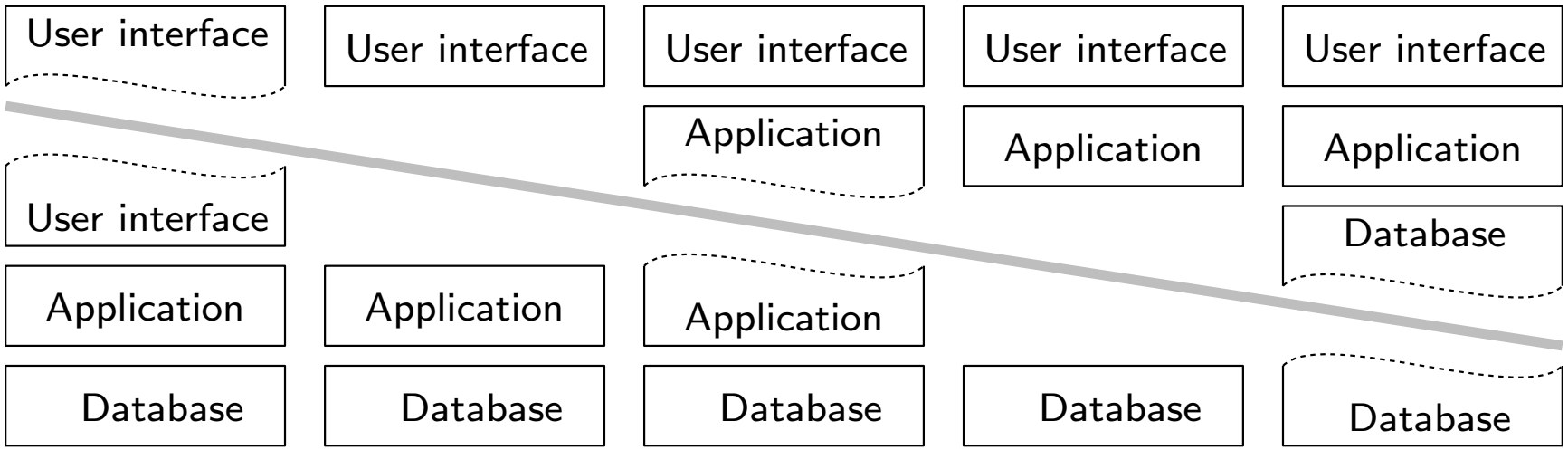
A typical server-client architecture (e.g., ERP systems, banking software) is split up into layers:

1. User-interface
2. Application
3. Database

Two-tier architecture:

- The two tiers are the client and the server.
- There is a cutting line somewhere between those three layers.

Two-tiered architecture



The two tendencies:

- *Fat client*: Put the weight to the client.
- *Thin client*: Put the weight to the server.

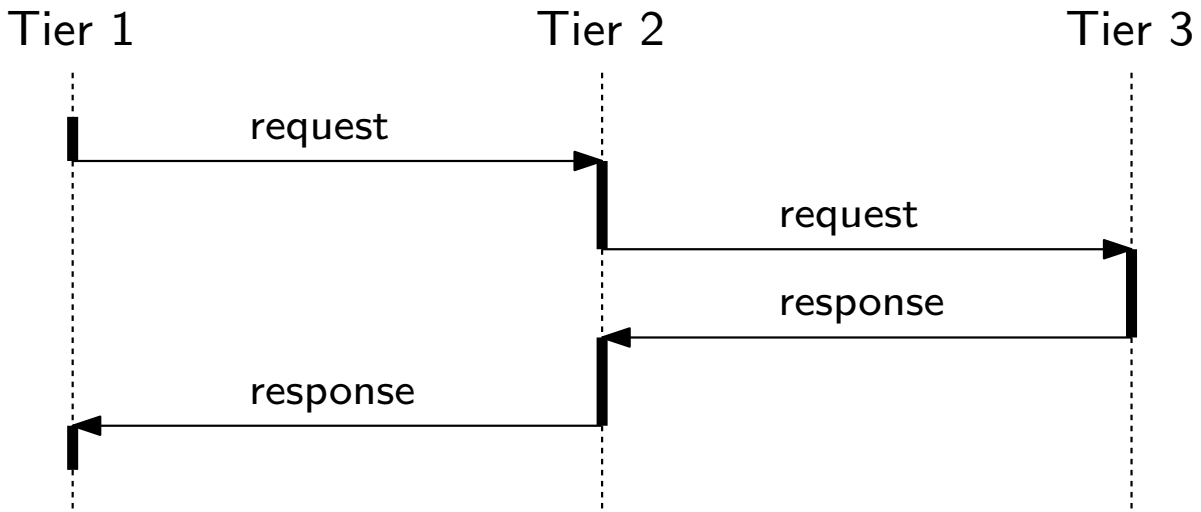
Fat versus thin clients:

- Complexity in administration, management, platform-dependency.
- Performance, richness of technology.

Three-tiered architecture

Typically the application server and database server is actually separated.

- Application server acts itself as client to the database server.
- Cascaded server-client relationship.



Reasons for three-tiered architecture:

- Looser coupling and higher scalability.
- The database is typically an *off-the shelf* component, whereas the application server might be develop in house.

CGI

CGI:

- Common Gateway Interface is a protocol[10] to pass HTTP requests on to applications.
- On request webserver launches application, passed request via environment variables, and reads result from stdout.

FastCGI:

- Webserver preforks a FastCGI process. (Why?)
- Communication to FastCGI process via Unix sockets, named pipes or TCP. Multiplexing allows concurrent request handling.

10 https://tools.ietf.org/html/rfc3875

Inter-Process Communication

Because of process memory isolation, communication between processes requires special means.

Most common techniques are:

- Shared memory
- Anonymous pipes
- Named pipes
- Unix sockets
- Localhost network communication

See `https://docs.python.org/3/library/ipc.html` for Python documentation on Inter-Process Communication.

Shared memory

- A process can request an *anonymous, shared* memory block from the operating system.
- When the process forks, parent and child have it both mapped in their virtual address space.
  - Concurrent access requires synchronization mechanisms!
- In C this is done by the function `mmap()`.
- In Python we have two data types for this purpose: `Value` and `Array` in the `multiprocessing` package.

```
1  import multiprocessing
2
3  def child(v, a):
4      v.value = 42
5      a[1] = 23
6
7  if __name__ == '__main__':
8      val = multiprocessing.Value('d', 0.0)
9      arr = multiprocessing.Array('i', range(10))
10     print(val.value, arr[:])
11
12     p = multiprocessing.Process(target=child, args=(val, arr))
13     p.start()
14     p.join()
15     print(val.value, arr[:])
```

Pipes

- A pipe[11] is a communication channel between two endpoints.
- In C the function `pipe()` creates a uni-directional channel; it returns a file descriptor for reading and one for writing.
- The very same function exists in Python as `os.pipe()`.

[11] Also called anonymous pipe or POSIX pipe.

Pipes in Python

- The `multiprocessing` package also provides `Pipe()`, which something different than POSIX pipes.
- It creates a duplex communication channel and returns a pair of connection objects on which `send()` and `recv()` can be called to send Python objects rather than raw bytes.

```
1  import multiprocessing
2
3  def child(conn):
4      print("recv:", conn.recv())
5
6  if __name__ == '__main__':
7      conn1, conn2 = multiprocessing.Pipe()
8      p = multiprocessing.Process(target=child, args=(conn2,))
9      p.start()
10
11     conn1.send(["hello there", 42.0])
12     p.join()
```

Queues in Python

- The `multiprocessing` package also provides `Queue()`, which allows for multiple producers and consumers.

```
1  import multiprocessing
2
3  def ch(q):
4      print("recv:", q.get())
5
6  if __name__ == '__main__':
7      q = multiprocessing.Queue()
8      for i in range(10, 13):
9          q.put([i, float(i)])
10
11     ps = [multiprocessing.Process(target=ch, args=(q,)) for _ in range(3)]
12     for p in ps:
13         p.start()
14
15     for p in ps:
16         p.join()
```

Unix Domain sockets

Using socket API:

- Unix sockets `AF_UNIX` (can) have better performance than the internet protocols `AF_INET`.
- Packet-oriented, but with sequence guarantees[12]: `SOCK_SEQPAKET` instead of `SOCK_DGRAM` or `SOCK_STREAM`.
- `socket.socketpair()` can create a pair of connected sockets using Unix Domain sockets, if available.
- Unix sockets can be bound to a file name.
  - The file system then acts as name space.

```
1  import socket, multiprocessing
2
3  def child(conn):
4      print("recv:", conn.recv(1024).decode())
5
6  if __name__ == "__main__":
7      conn1, conn2 = socket.socketpair()
8      p = multiprocessing.Process(target=child, args=(conn2,))
9      p.start()
10
11     conn1.send("hey there!".encode())
12     p.join()
```

12 All packets arrive in order, unlike for UDP.

Named pipes

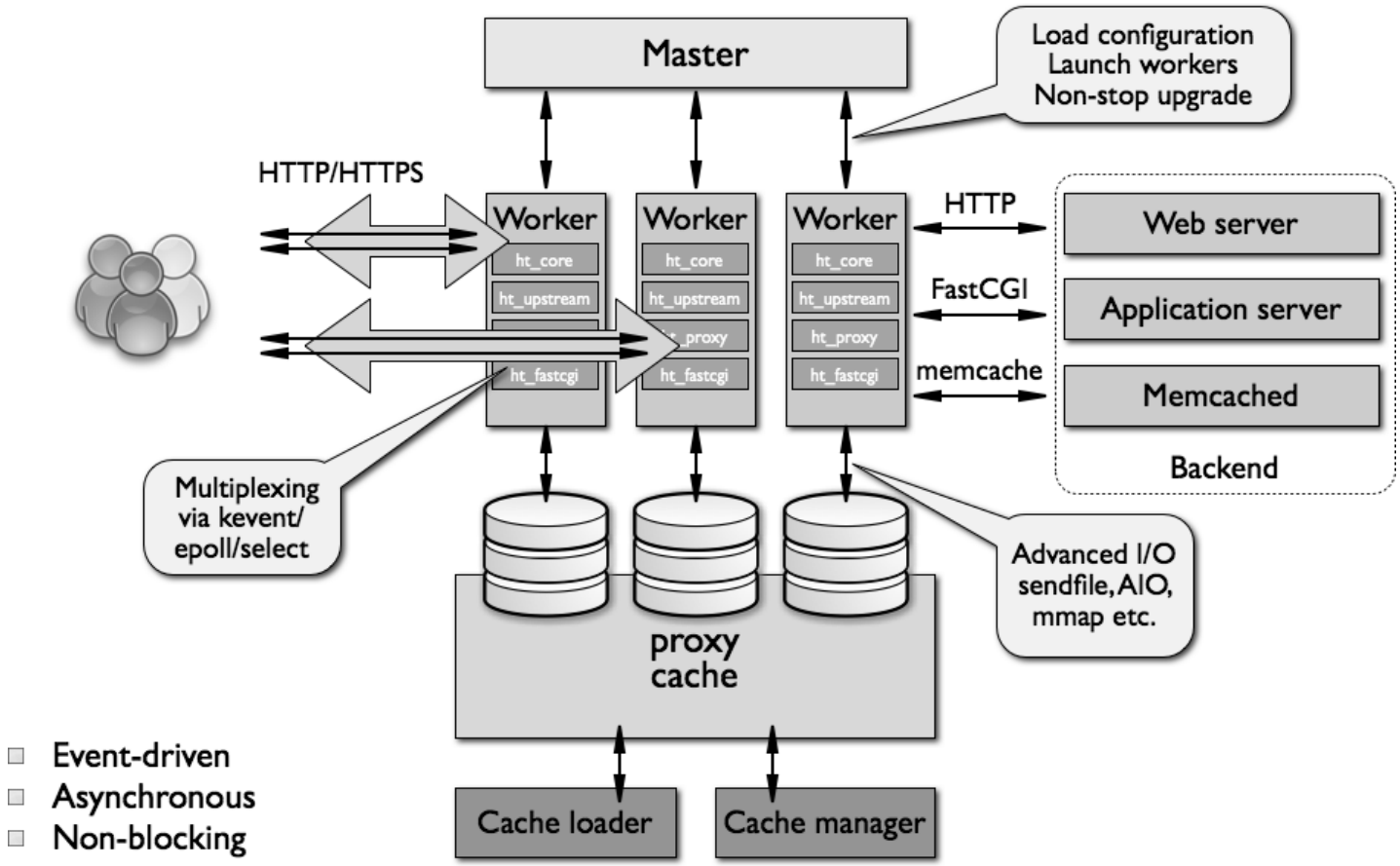
A named pipe is a FIFO special file:

- Like an (anonymous) pipe, but bound to a file name.
- Once created with `mkfifo` it can be opened with ordinary file I/O operations. The data, however, is not really written to a file system. The file system is only used as name space.
- Reading blocks until another process writes to it. (Unless non-blocking I/O is used.)
- In C there is a function `mkfifo()` and in Python there is `os.mkfifo()`.

```
mkfifo foo.fifo
ls -l foo.fifo

# Shell 1
cat - foo.fifo
# Shell 2
cat foo.fifo
```

NGINX architecture



- Event-driven, asynchronous, non-blocking
- Preforked worker pool and shared-memory IPC
- Three-tiered architecture, FastCGI