12

Contents

- Lab related
  - Thread
  - I/O multiplexing
  - Proxy architecture
  - Race condition
  - Cache implementation

std::thread

- std::thread is easier to use than pthread
  - Pthread job is limited to "void *job(void*) which is not easy to use"
  - However, you can pass any "callable" object to contstruct a std::thread object
- Usage:
  - std::thread my_t(funcname, param1, param2...)
  - my_t.detach() or my_t.join()
- Need to add "-lpthread" flag when compiling
  - Both pthread and std::thread requires the flag

Example

```cpp
#include <iostream>
#include <thread>
#include <string>

struct FuncObj
{
    std::string name;
    FuncObj(const std::string &n):name(n){}
    void operator()(int a, int b)
    {
        std::cout<<a<<" "<<b<<" from "<<name<<std::endl;
    }
};

void normal_func(const std::string &name, int a, int b)
{
    std::cout<<a<<" "<<b<<" from "<<name<<std::endl;
}

int main()
{
    std::string name = "main";
    std::thread t1(FuncObj("obj"), 1, 2);
    std::thread t2(normal_func, "nor",1, 2);
    std::thread t3([&](int a, int b){
                std::cout<<a<<" "<<b<<" from "<<name<<std::endl;
            }, 1, 2);
    t1.join();t2.join();t3.join();
}
```

Join and Detach

- There should be someone revoke the resource from a **terminated** thread
  - Two primitives: **Join** and **Detach**
    - **std::thread::join** and **std::thread::detach** in C++ thread support library
    - **pthread_join** and **pthread_detach** in pthread library
- Function *main* call ***A*.Join()**, then *main* will be blocked until ***A*** is terminated. The resource will be deallocated by *main* afterwards.
- Function *main* call ***A*.Detach()**, then object ***A*** will have no relationship with the detached thread. The system will do free the resource after the detached thread terminated, automatically.
- Manage threads carefully!

Notice

- In pthread library, threads are referenced by *pthread_t* variable
- In std::thread, threads are referenced by **std::thread** object
- After **Detach** and **Join**, the relationship between the thread and the "handle" is deleted.
- **C++ runtime will destruct object implictly**
  - Thread **t** is referenced by the object *obj_t*
  - When **t**'s destructor is called, **std::terminate** will be called **IF**:
    - **t** is neither detached nor joined

poll – wait an event on a descriptor

- **poll**() performs a similar task to **select**: it waits for one of a set of file descriptors to become ready to perform I/O.
- User pass an array of ***struct pollfd*** to the **poll** function.
  - No macro or bit mask operations.
- Easy to extend to C++.
- Support timeout.

- Example:

```
class PollMgr{
    public:
        void Insert(pollfd, callback, mode);
        void Stop();
        void Run();
    private:
        int timeout;
        … // pollfd list and callback
            // list
};
```

Easy to do I/O multiplexing

- Just use PollMgr in the previous silde.
- Set up the manager by insert pollfd and callback function into it.
  - The callback function will be called when the fd is readable/writeable.
- Can still insert new pollfd in to a running manager
  - Simple solution: Takes effect after next "timeout".
  - Other solution: use thread

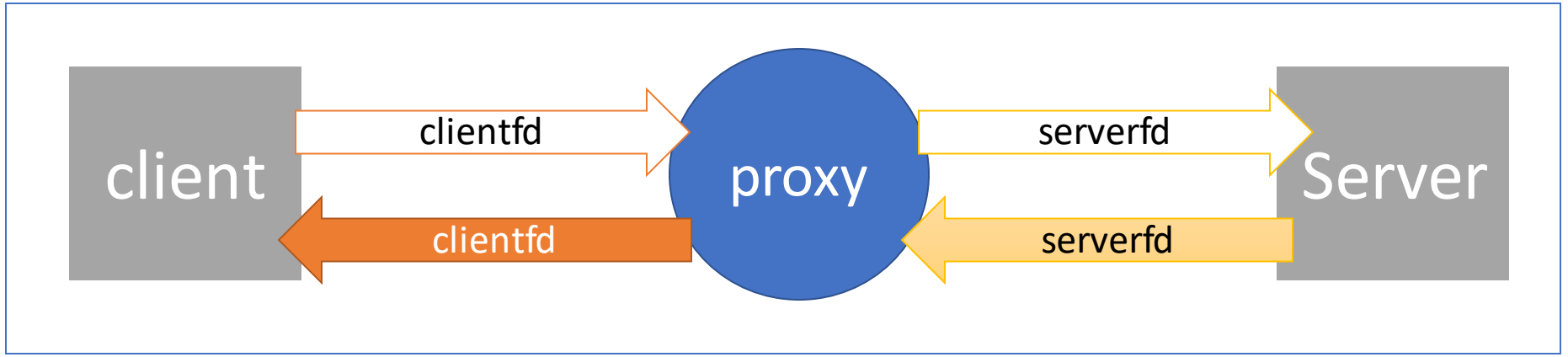
Wireshark

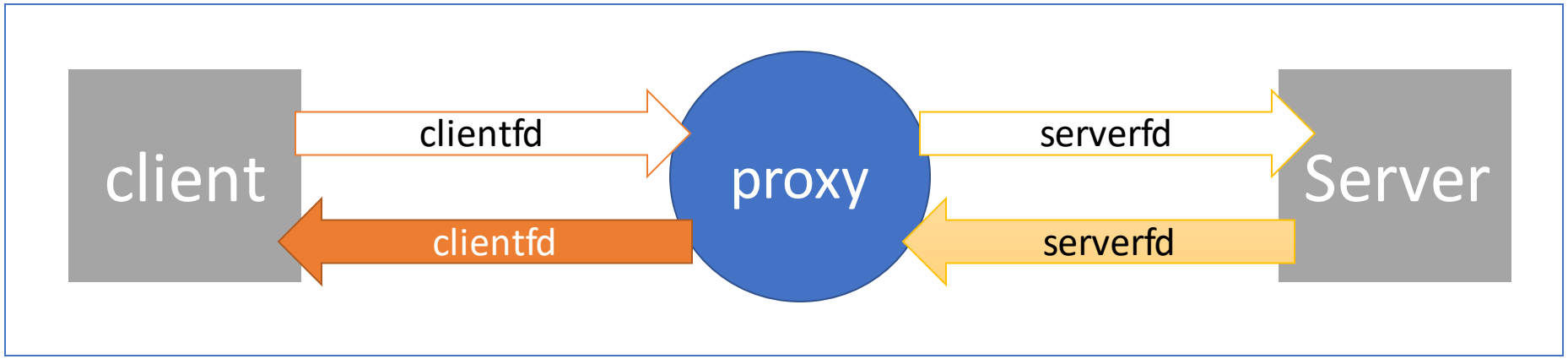
- A tool to capture packets
- Packet format sometimes is weird
  - GeT http://baidu.com/index.html HTTP/1.1
  - gEt baidu.com HTTP/1.1
- Use wireshark to see the content of the packet
  - Help you debug
- Installation: $ sudo apt install wireshark
- Usage: $ wireshark

Proxy architecture

- Only I/O multiplexing
- Proxy should be able to support multiple clients
  - Use I/O multiplexing
  - Need to know which server the client sends to. --> use a "map" to store
- Proxy should be able to forward packets between client and server
  - Need to block on both "clientfd" and "serverfd" and wait for input from each side.
  - Use I/O multiplexing.

Proxy architecture



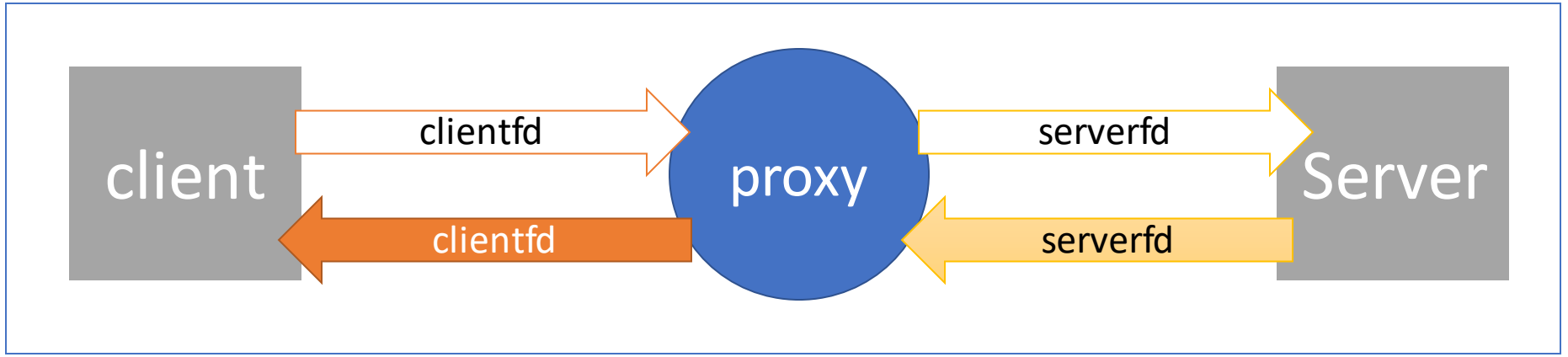


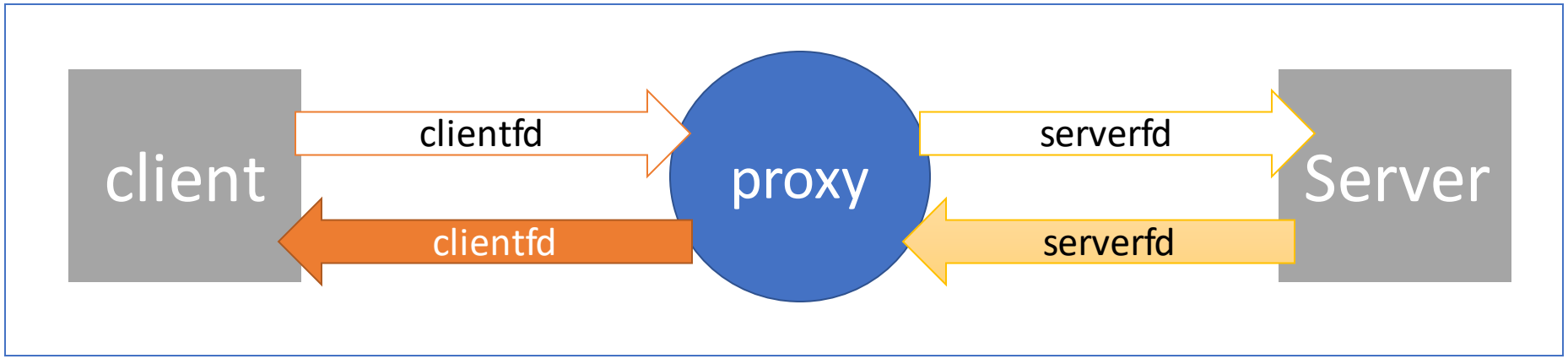
Proxy architecture

- Thread + Process
- Proxy should be able to support multiple clients
  - Use multiple child processes
- Proxy should be able to forward packets between client and server
  - Use two thread to manage clientfd and serverfd.

- Problem: Hard to implement global cache
  - But each client can have its own local cache, it still works

Proxy architecture

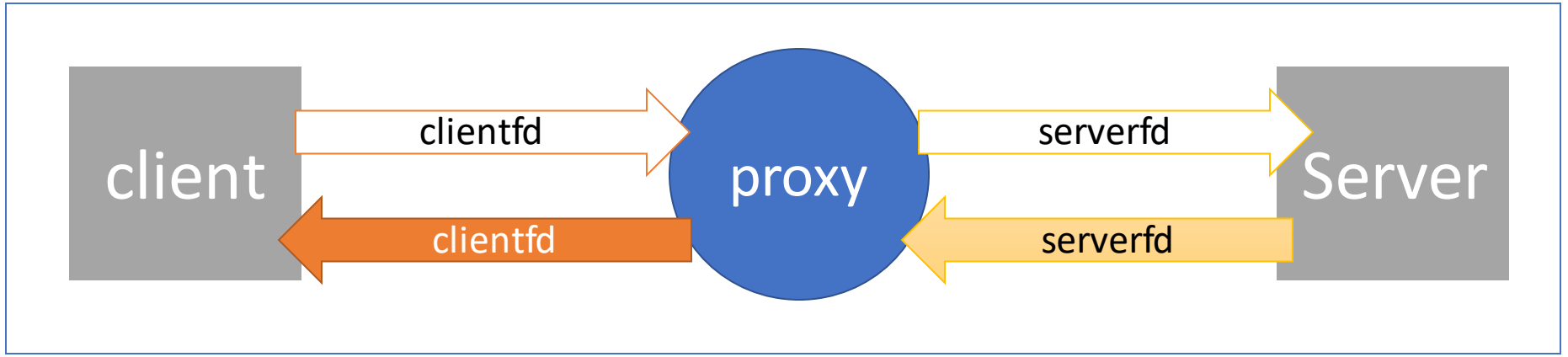


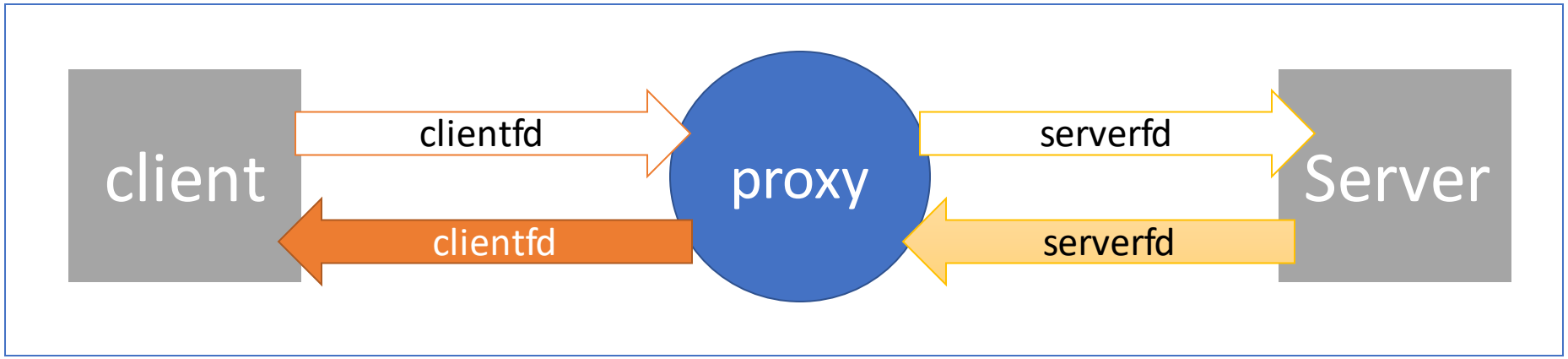


Proxy architecture

- Thread + I/O multiplexing
- Proxy should be able to support multiple clients
  - Use threads
- Proxy should be able to forward packets between client and server
  - Need to block on both "clientfd" and "serverfd" and wait for input from each side.
  - Use I/O multiplexing.

Proxy architecture





Race condition

- Two threads wants to modify one shared variable
  - Read to local --> modify --> write back to memory
- Problem:
  - There are two threads A and B executes the same code "a[i]++;"
  - a[i] is 0 initially
  - Operation on a[i] take 3 steps: *read, modify, writeback*, and each step is atomic

Race condition – problem 2

- Two threads A and B, executing "a[i]++" for 1000 times.
  - a[i] = 0, initially.
  - Still take 3 atomic step: read, modify, writeback
- Question: what's the **range** of possible value of a[i] after both A and B finish their execution.

Avoid Race condition

- P/V operation. -- On the text book.
- Mutexs
  - pthread_mutex
  - Mutexes in C++ thread support library

pthread_mutex

- Usage:
  - pthread_mutex_init(&m, NULL): initialize a mutex called **m**
  - pthread_mutex_lock(&m): lock the mutex **m**
  - pthread_mutex_unlock(&m): unlock the mutex **m**
- Can protect data or code
  - Only one thread can access the code in the "locked" area.
  - The "locked" area is called "**critical section**"

C++ mutex

- **std::mutex** and **std::unique_lock<std::mutex>**
- Lock the scope when the **unique_lock** object is constructed. Unlock the scope when it's destructed.

```
Void function(){
    Safe code...;
    std::mutex m1;
    { // new scope for critical code
        std::unique_lock<std::mutex> lock(m);
        Racing code...;
    } // critical section end
    Other safe code...;
}
```

Cache system

- Implement as a module! Separate from your proxy code.
    - Code style!
- Global cache may have race conditions!
    - Multiple threads wants to read/write the cache simulteanously.
- Use mutex to lock the critical section.
    - Such as LRU eviction or "cache add" function.