# Trickle*: A PSP File Sharing Protocol

Aniruddha Deb

September 2022

## Contents



*The opposite of Torrent :)

# 1 Design

The design for trickle was guided by three core principles:

1. *Speed:* The bandwidth should be the bottleneck in file downloads, not the server/client connections, or the software layer
2. *Scalability:* Given a large enough server cache and ample bandwidth/processing power at the server, the server should be able to concurrently service a large number of clients
3. *Reliability:* Irrespective of the Control layer protocol used (UDP or TCP), control messages should achieve their end purpose. This is different from stating all control messages should reach their target: if we don't receive a control message we're expecting, we re-request it until we get it.

More details on how these principles were incorporated are given in the following sections

## 1.1 Design Decisions

Following from above, *C++* was chosen as the language of choice because of it's speed and closeness to the system. Sockets are implemented as syscalls in C, using the POSIX socket library (as a result, this implementation doesn't work on windows). Interfaces were designed in C++ to abstract away the underlying C code for memory safety and portability (more on this in the Code Layout section).

To ensure scalability, *We have to abandon the premise that one client will be serviced by one thread at the server*. Upon having a large number of threads, the OS would spend more time scheduling the threads rather than actually running the threads. This is a well-known, known in systems literature as the C10K problem (the ability to concurrently handle 10,000 clients at once).

The solution to this problem is to use a design pattern known as the Reactor Pattern: rather than polling the sockets and waiting till we can read/write on them, we assign this task to the kernel. The kernel would then notify us of when a socket can be read to / written from, and we then read/write to the socket at this point.

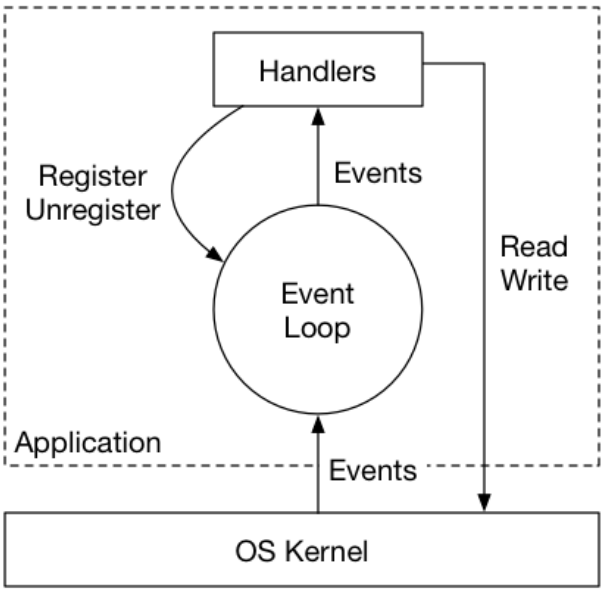


Figure 1: The Reactor Design Pattern (Credits: TeskaLabs)

The Reactor design pattern is implemented in our code using the kqueue library. kqueue is the default event queue on MacOS/BSD (my platform), and for other platforms, libkqueue provides a compatiability layer over the underlying event queue library used by the kernel (Linux uses epoll, Windows uses IOCP)

For our third requirement, we register a timer event with our event queue. The timer event triggers a callback every second, and in this callback, we check if there are any stale chunk requests in our buffers (which haven't been responded to by the server). If there are, we clear these requests out and send them again, assuming that they have dropped.

## 1.2 Design

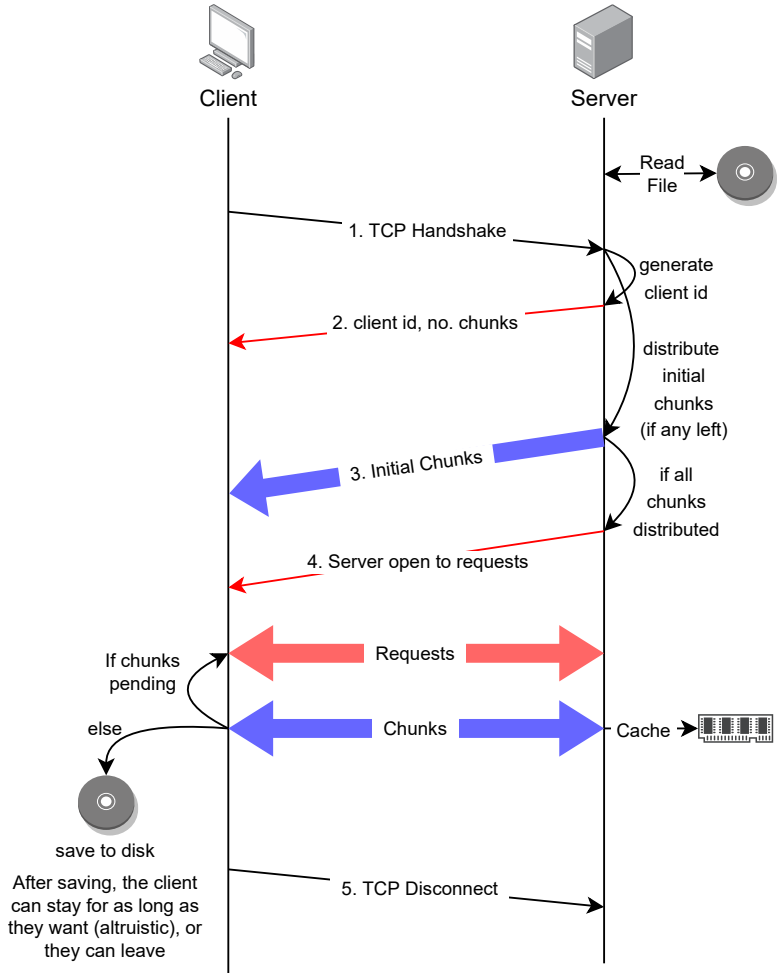


Figure 2: Flow diagram of the trickle protocol (Red - Control layer, Blue - Data layer)

Trickle uses two layers: a control layer for sending messages, and a data layer for sending chunks. The protocols used for them vary: Part 1 of the assignment implements the control layer in UDP, and the data layer in TCP, while part 2 does the opposite. We refer to different implementations by *the protocol used for the control layer*. Hence `server_udp` represents a server using UDP for control and TCP for data, and `client_tcp` represents a client using TCP for control and UDP for data.

**The server is implemented Using Only Two Sockets**, one TCP and one UDP, located at the same port of the host machine. Clients are also implemented in the same manner, with both TCP and UDP at the same port. This makes sending messages easier, and keeps the client communicated. The client doesn't need to randomly choose a communication port, and the server doesn't need to

keep track of the client addresses at it's various ports. The asynchronous event-loop based structure allows the server to handle multiple clients at once: reads and writes on socket file descriptors are non-blocking, and kqueue informs us of when we can read/write from the socket.

**Note that both the client and server are asynchronous**: The client does not need to wait for a response to a chunk request to initiate the next chunk request, and neither does the server have to wait for a particular client to reply to a chunk to initiate more chunk requests. This allows for a very fast transfer of data over the network, leaving the burden of sequencing and tabulating the incoming data to the client machine.

## 1.3 Protocol

Two types of messages, Control messages and File Chunks are transferred in the control and data layers respectively. The structure of these chunks is given in fig. 3.

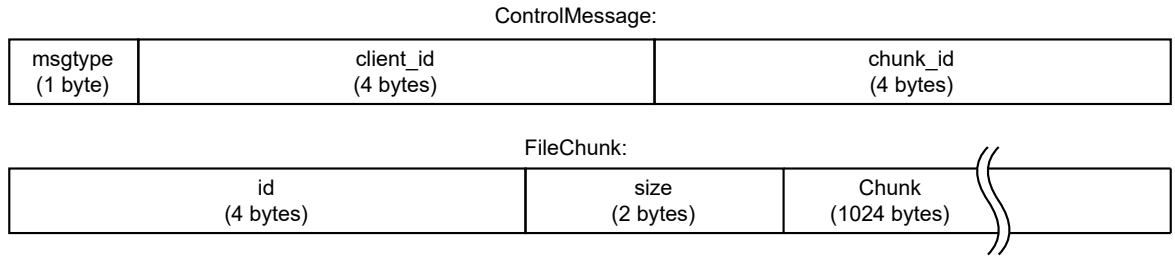


Figure 3: Messages used in trickle: ControlMessage and FileChunk

The control messages can take the following values:

| msgtype | client_id | chunk_id |
|---|---|---|
| REQ | ⟨ client id ⟩ | ⟨ chunk id ⟩ |
| OPEN | - | - |
| REG | ⟨ client id ⟩ | ⟨ num chunks ⟩ |

The FileChunk has 4 bytes for the id, 2 bytes for the size of the data in the chunk and 1024 bytes for the data itself. Note that a chunk can have less than 1024 bytes, if it's the last chunk in the file. For proper reconstruction of the file, the size field is important, as it signifies how many bytes of the data are meaningful.

## 1.4 Code Layout

To accomodate the different implementations, a modular design had to be used: we define three major interfaces, **Client** (representing the client object), **Server** (representing the server object) and **ClientConnection** (representing a connection to a single client on the server side). An overview is given in fig. 4.

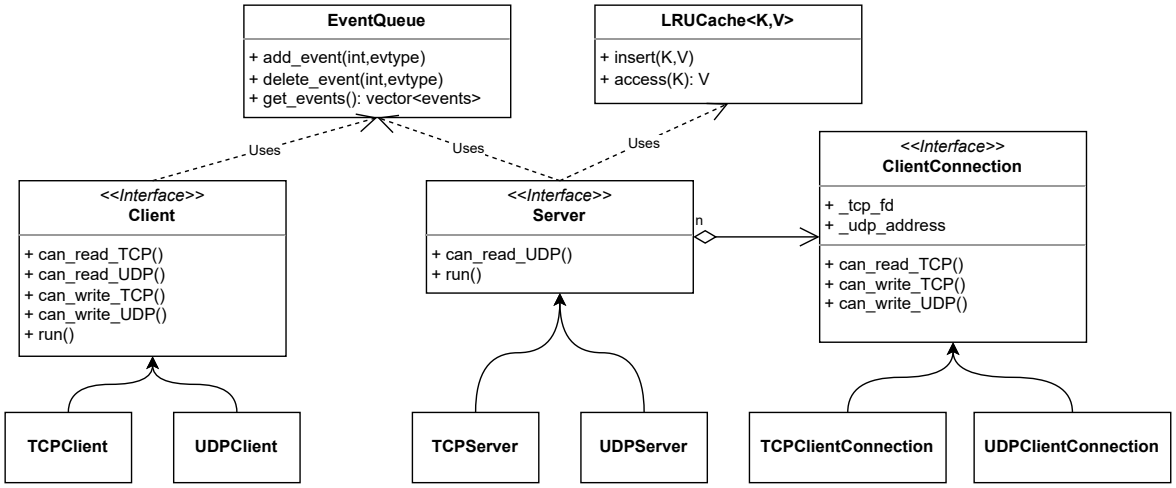


Figure 4: UML diagram of the major interfaces and their implementations

The interfaces are defined as hpp header files, with code for common implementations defined in the header file itself. The implementation of the specific methods is defined in the `*_tcp.cpp` and `*_udp.cpp` files. The choice of choosing a TCP or a UDP implementation is made at compile time, by passing the `PROTO=TCP` flag to the Makefile.

## 1.5 Handling Packet Drops

Packet drops are solely handled by the client side: in case a particular message/chunk drops, the client has a timer that alerts it every second. It will send a request to the server containing the header of the particular message it's expecting for OPEN and REG requests, and will resend the chunk request in case of chunk drops. The timeout for chunk drops is set to be 5 seconds on the client side, and for OPEN and REG misses, is 1 second.

## 1.6 Installation and Execution

NOTE

The code was tested on Mac (MacOS 10.15.6, g++ v11.2.3) and Linux (Manjaro 21.3, g++ v12.1.1, libkqueue v2.6.1). This program is not compatiable with Windows, and also requires building and installing libkqueue from the GitHub source if on Linux (`apt-get`/`pacman` have outdated versions, which have bugs)

The files are structured as follows:

```
trickle.zip
├── trickle.pdf
├── trickle.sh
└── trickle_proj
    ├── Makefile
    ├── res
    │   └── A2_small_file.txt
    └── src
        ├── client.hpp
        ├── client_connection.hpp
        ├── client_connection_tcp.cpp
        ├── client_connection_udp.cpp
        ├── client_tcp.cpp
        ├── client_udp.cpp
        ├── clientmgr.cpp
        ├── event_queue.hpp
        ├── lru_cache.hpp
        ├── protocol.hpp
        ├── server.cpp
        ├── server.hpp
        ├── server_tcp.cpp
        └── server_udp.cpp
```

The bundled shell script compiles and then runs the code located in `trickle_proj`. The executables are generated in `trickle_proj/bin`, and the code is run by the script. The `md5` utility is used to generate a md5sum of the generated files, which are stored in `trickle_proj/output`.

# 2 Analysis

## 2.1 RTT averages

The average RTT across all chunks, across all clients for the UDP implementation (Part 1) was **33423** $\mu$**s**. For the TCP implementation (Part 2), the value was **21011** $\mu$**s**. This is when both clients request chunks sequentially, from 0 to $n-1$, and chunks are distributed in a block based manner to all clients (that is, client 0 receives 0..m-1 chunks, client 1 receives 1..2m-1, and so on).

The value for the TCP implementation on average is smaller. This is as expected, as the bulk of the data (that is, the chunks) are transferred over UDP. UDP has no rate limit and no error correction/packet drop protection, hence the client and server can fire away as many chunks as they want over a short period of time. This may lead to congestion (as can be seen in the average RTT across time), but for the small number of clients and chunks here, it is much faster.

## 2.2 RTT averages across clients

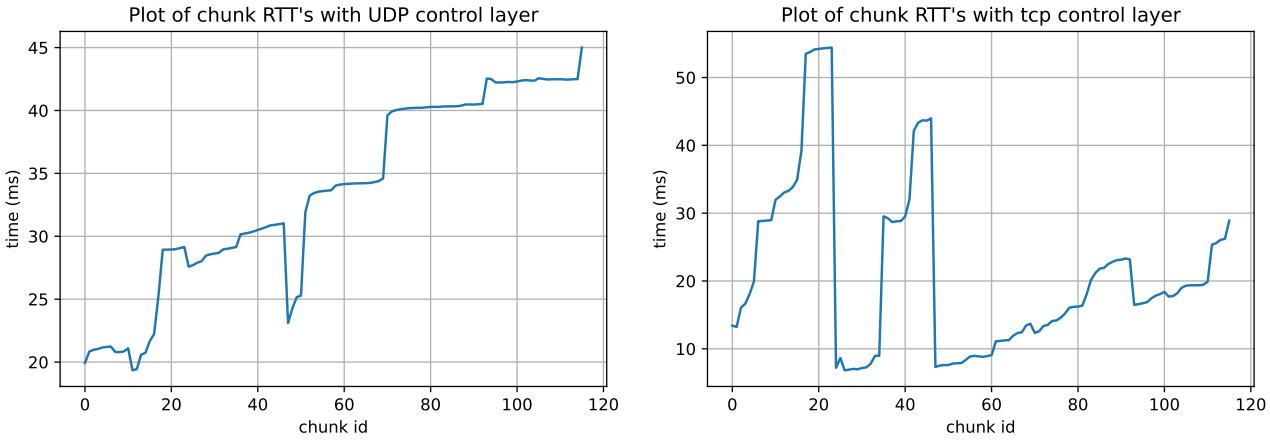


Figure 5: RTT averages for each chunk across clients

- For the UDP implementation, since chunks are transferred via TCP, we see a monotonic rise in packet request times as we approach the end. As we get to the last chunk, there are more senders of the chunk than receivers, and this generates a lot of congestion on the network. Since TCP has to be reliable and can't drop packets, the time to transmit over this congested network increases.
- For the TCP implementation, since chunks are transferred over UDP, there are spikes at chunks 20 and 40: this is because of high demand for a particular chunk. Only one client would have that chunk, and all the four other clients would be demanding that chunk from that particular client, hence the time would increase. Cache misses are also more frequent towards the starting, causing the initial request times to be high.

## 2.3 Load Benchmark

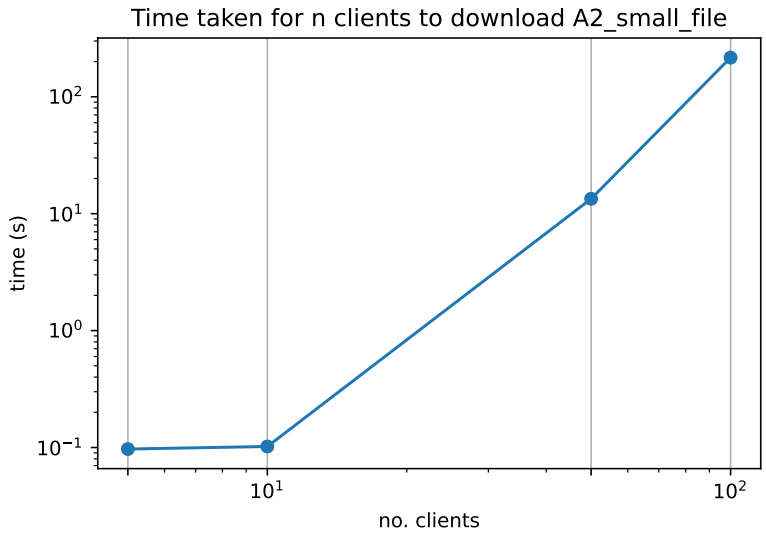


Figure 6: load statistics with cache size 100. Note that time is recorded from the point the last chunk leaves the server to the point where the last client saves it's file.

The load benchmarks are as expected: due to the presence of the central server, as the number of clients increases, the time increases as well (the increase upto 10 clients is not noticeable on a log-scale, due to the fact that there are not enough clients/data being exchanged to generate a bottleneck). For every order of magnitude increase of the number of clients, the time taken increases by three orders of magnitude.

Note that this increase may also be due to simulation: on my machine, the process simulating the clients took over 95% of all available resources for $n = 100$, leaving only 5% for the server (each client was simulated using a different thread). A rigorous load benchmark would involve running the server on a network, and having a large set of clients from another machine connect to it.

## 2.4 Cache Size Benchmark

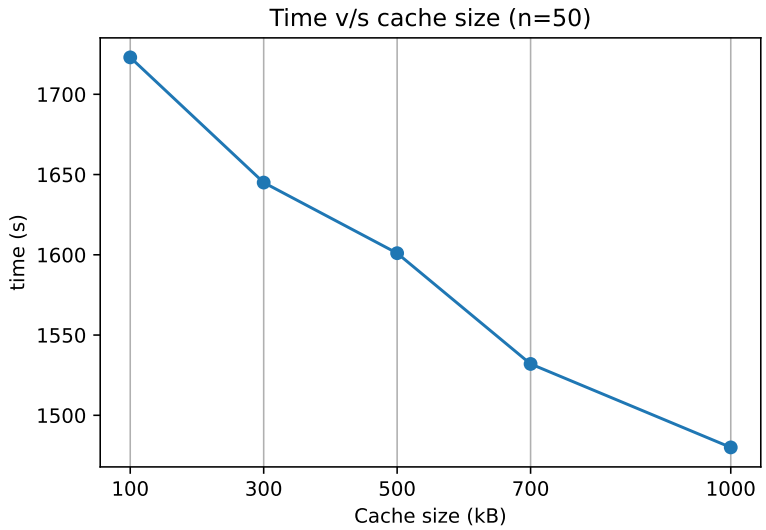


Figure 7: Time vs cache size plot for 50 clients to download A2_large_file.txt

Simulating this with a 100 clients on my machine was not possible, as it has only 8GB of memory, and each file takes up 100MB. Since we are not intermittently storing chunks to disk and everything resides in memory, we would need $\sim$ 10GB of memory to simulate this.

Fewer cache sizes as well were simulated: I could only simulate a subset of the initially requested cache sizes, as the time to run the tests was very high (nearing almost 30 min for the longest one). However, the results are in agreement with what is to be expected: *there is a decrease in the total time with an increase in the cache size*. This is because access from cache is faster than going across the network, and a larger cache size implies fewer cache misses.

The decrease is only marginal: going from 30 minutes to around 24 minutes, because the cache size is still a very small fraction of the total number of chunks. There are $\sim$ 103,000 chunks in the large file, giving us a cache to file ratio of 1:1000 to 1:100. This is too large to have a significant reduction in the time.

There is also little to no **Cache Overlap**: Due to the sequential nature of requests and chunk distribution, at any time, the majority of chunks requested would form a window, whose length would be approximately the number of chunks distributed to any one client. Each client gets $\sim$ 2000 chunks in this scenario, while the maximum number of chunks in the cache is only 1000. Hence, by the time we get to a run of chunks that have been requested previously, the cache would contain little to no chunks from that sequence, as they would all have been evicted.

## 2.5 Randomized Client Request Sequence Benchmark

On randomizing the client requests, The process took marginally more time (103 ms compared to 97 ms). This was due to the small number of clients and the speed of the server. However, the average RTT per chunk changed drastically:

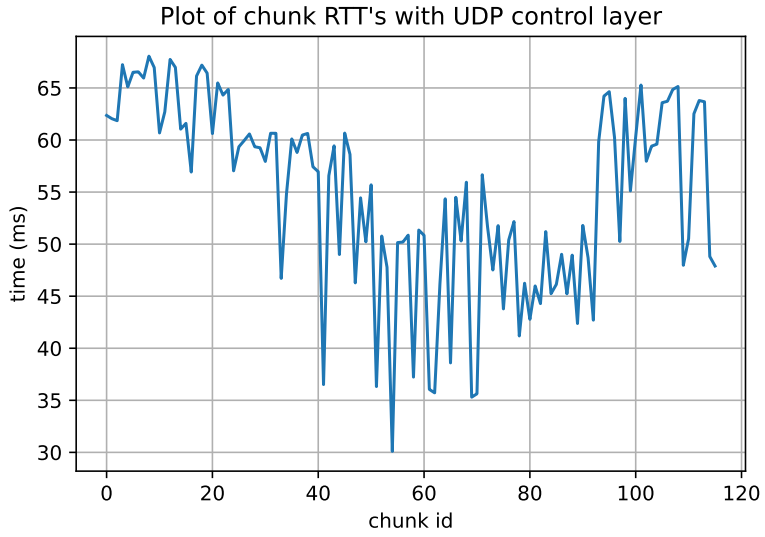


Figure 8: Average chunk response times when requests are random

The times are still low, due to the cache size to file ratio, which is 100:116 $\sim$ 0.85. This is a high number, and the random requests have only a 15% chance of generating a cache miss. As the cache size to file size ratio grows smaller, We expect a randomized request sequence to generate more cache misses, causing the time to shoot up drastically compared to a sequential sequence, where the number of cache hits would be more.