Trickle*: A PSP File Sharing Protocol

(A report for Assignment 2 of COL334)

Aniruddha Deb
2020CS10869

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.

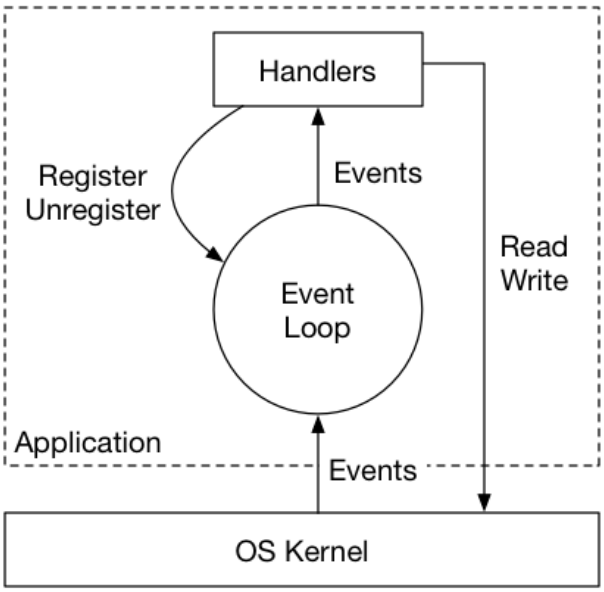

Figure 1: The Reactor Design Pattern (Credits: TeskaLabs)

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

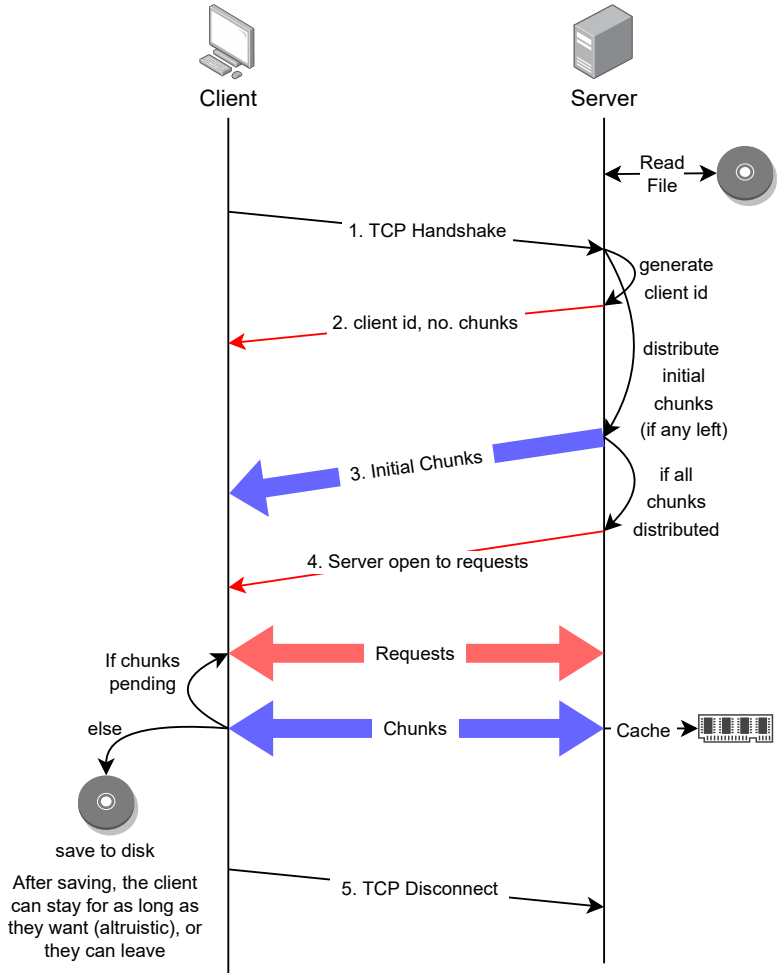

Figure 2: Flow diagram of the trickle protocol (Red - Control layer, Blue - Data layer)

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.

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.

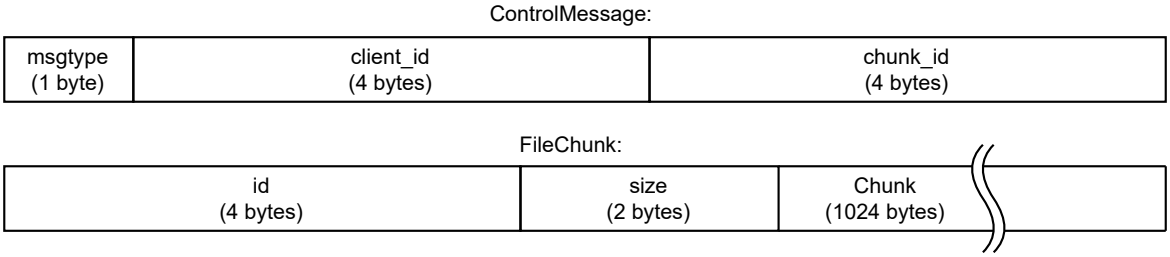

Figure 3: Messages used in trickle: ControlMessage and FileChunk

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.

1.3 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.

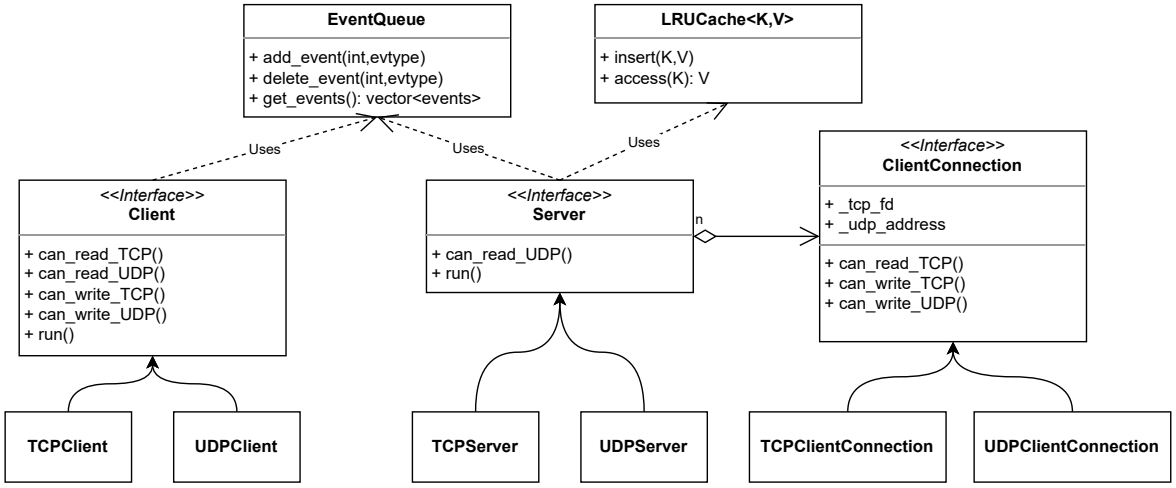

Figure 4: UML diagram of the major interfaces and their implementations

1.4 Handling Packet Drops

1.5 Client and Server Pseudocode

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.

2 Analysis

2.1 RTT averages

RTT avg UDP: 33423, RTT avg TCP: 21011

2.2 RTT averages across clients

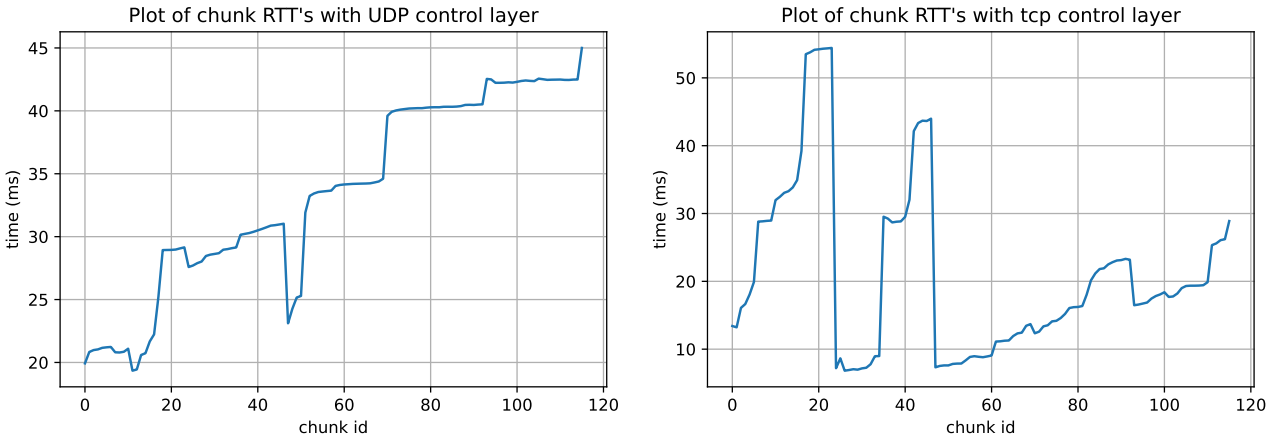

Figure 5: RTT averages for each chunk across clients

2.3 Load Benchmarks

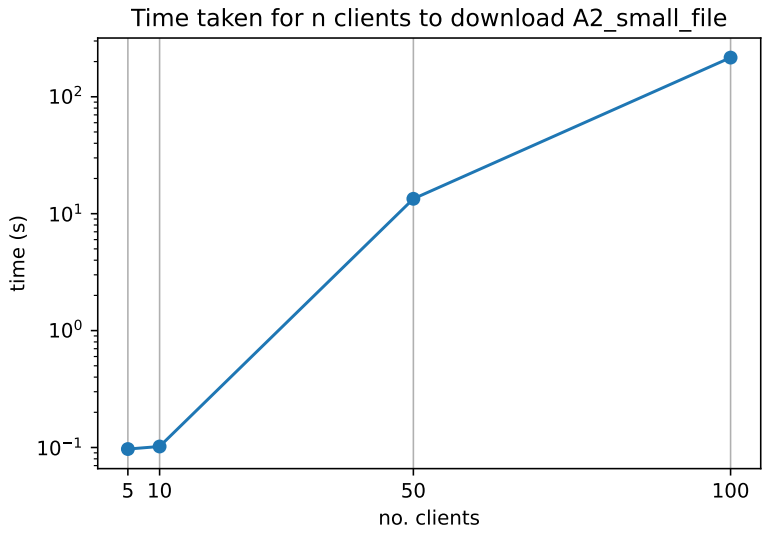

Figure 6: Load statistics with cache size 100 (note that the y-axis is logarithmic)

2.4 Cache Size Benchmarks

2.5 Client Request Sequence Benchmarks

55 ms mean

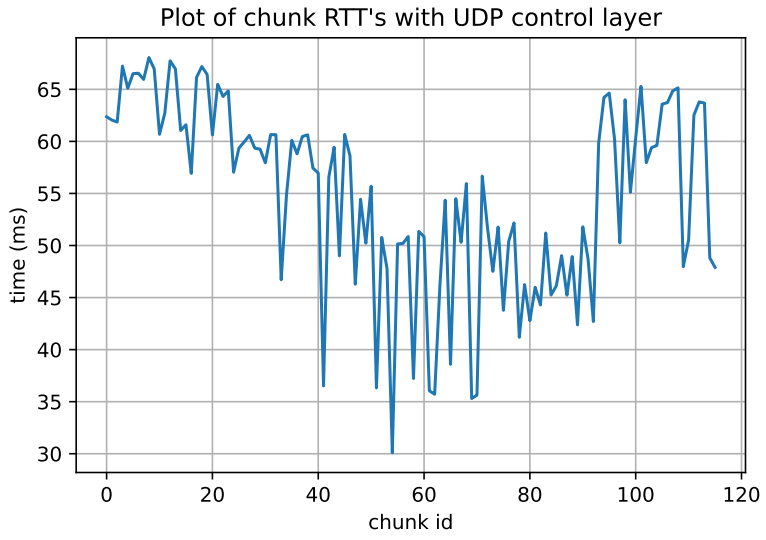

Figure 7: Average chunk response times when requests are random

3 Food for Thought