Application-layer protocol for highly-available file server downloads

1. Abstract

Existing methods for attaining high availability for networking applications centres around transport and network layer implementation. We explore a potential application layer protocol for unidirectional downloads of static resources as in [1]. Whilst the implementation is incomplete, the more restricted case where implementation is complete shows promising results.

2. Introduction

In this report, we describe the development and operation of a system intended to expand upon the work of Snoeren et al. [1] in achieving a highly available service for unidirectional static file transfers. Snoeren et al. describe two methods for connection migration: migrating the TCP connection at the transport-layer using one of several proposed TCP extensions (including [6]); or having the backup server inform the client that the communicating server will change and having the client reinitialize the connection. We explore an implementation of the second unexplored idea at the application-layer to support mobility between systems that may not have the necessary TCP extensions for existing transport-layer protocols.

3. Related Work

There are several existing approaches towards providing highly available networking applications. One solution is to provide fault tolerance at the transport-level protocol, typically TCP. This is the approach of FT-TCP [2]and ST-TCP [3]. The idea is never to end the TCP connection and to instead switch the server providing responses. This allows the failover to be invisible to the client.

In these approaches, failover is to a backup server, which is either a single or several backup servers. Commonly, the backup servers are co–located with the primary server to prevent server–resurrection issues, causing duplicate IP addresses or multiple servers communicating with the client. The backup server detects a failure in the primary server, and then the connection is migrated to the backup server, and the TCP stream continues. A variety of mechanisms for IP takeover exist for co-located servers [5]. Changing the primary server can be handled with a switch that suppresses packets from all servers except for the primary server. When the primary server fails, a new primary server is chosen and now passes packets outside the local network. The former primary is suppressed to prevent resurrection issues.

Most connections have a state attached; the server code maintains some information on the communication that needs to be replicated to any backup server to allow a transparent failover. One approach to replicating is to have the backup servers receive copies of all incoming packets via a switch and drop any packets from the backup servers; this is a straightforward extension to the failover mechanism described above. The state can also be maintained by storing it on a shared storage system before replying. When a failure occurs, the backup server can read the state from the dependable storage system. Replies to clients as well as other data maintained can all be retrieved to minimise the failover time depending on the exact nature of the connection fault.

Both approaches to maintaining the state have drawbacks; replicating the communications necessitates the development of an often non-trivial leader/follower protocol to detect primary server faults and elect a new primary server [5]. A shared storage mechanism introduces another component that can fail and trades one set of problems for another.

Regardless of the specific failover mechanism, transparency typically necessitates that servers are co-located to allow a switch to control failover. Transparent replication can create many design challenges that significantly increase the system's complexity and may not be wholly desirable for developers [4]. These methods may also involve modifying the TCP/IP stack on all participating devices, including intermediate relays in different administrative regions, making them impractical for industrial use.

Other approaches towards high availability allow the client to be a participant in the process, defining an application or transport-layer protocol that encapsulates the connection to provide continuity of service. Whilst this cannot be implemented by only modifying the server code, the necessity to alter intermediate relays or the TCP/IP stack is eliminated, making implementation more feasible and is, therefore, the approach we have selected.

This approach's flaw is that in a general case, the synchronisation of application-level and transport-level state becomes much more complex. Whilst addressing the general case may be of importance, there are many more restricted cases where this problem can be simplified and, whilst less appealing for research, the narrower cases are often more relevant to industrial work [4]. Snoeren et al. [1] addresses the issues of static file downloads and presents an intuitive application-level synchronisation protocol where the current state of a server's connections are disseminated at a frequent interval, and information expires after a period. The claim is that this synchronisation mechanism offers sufficient robustness provided values for frequency and expiry that minimise the probability of no other server having knowledge of a connection.

The static nature of the files serviced means that the only data to synchronise between servers is the file being downloaded and not the current state of a dynamic back-end, as would be the case for many modern web-pages which rely on dynamic content. This limits the protocol's utility but still addresses valuable points (e.g. a Content Delivery Network).

When server failure has occurred, several mechanisms can allow communication to continue with a new server. A backup server takes over the TCP connection and messages the client that the communicating server has failed and that the connection endpoint will change. If multiple servers attempt this, the one that receives a transport-layer acknowledgement is the accepted server. This migration can be wholly implemented at the transport-layer. The stream continues based on transport-layer information (i.e. sequence number).

Alternatively, when a server fails, the backup server can inform the clients that its communicating server is dead and needs to change. The client can then terminate the existing connections and initiate to a new server and continue to receive the stream from where the previous server left off. If reconnection is to the contacting server, this can aid load balancing across servers — busier servers should have a longer delay before contacted failed clients. This is the approach we have selected for use as it can be implemented wholly at the application-layer simplifying implementation and increasing portability.

The final aspect to consider is the detection of server failure. This is implemented as a health monitor of some description. The server or the client may implement health monitoring. Client-side implementation is trivial — if the client expects a packet and does not receive one within an interval of time, it determines the server has failed. This approach is only applicable when the client chooses and controls the migration as in M-TCP [7]. The drawback of the client controlling migration and server selection is that the client lacks knowledge of server-load and therefore is more likely to load servers unevenly.

A better option for load-balancing is to have the servers control the migration and server-selection. In many cases, the servers have adequate knowledge of their peers to offer migration to servers with lower load. Detection of failure by the server can be handled by a distinct heartbeat protocol [3]. A server sends periodic packets to its peers; this is the servers heartbeat. Much like a person, if the heartbeat has stopped, the server is assumed dead. If the fault has not occurred at the server, but instead the inter-server communication, the likelihood of false reports can be reduced. Moreover, for many applications, over-eager reporting of death often has a negligible impact on performance since failover time can be minimised with a more frequent heartbeat [3][1]. This is the approach we have selected.

A final option for co-located servers that use active-replication is to detect when the primary and backup differ in response time or packet contents. Avoiding using time-outs to detect errors means detection time is potentially shorter [8], and this may detect other faults than server death or connection failure as the server's response can also be compared. Though, this adds the complication of erroneous operation on any backup server. Additional hardware costs to ensure backup servers operate at similar speeds and the co-location requirement add a significant cost to this approach.

4. Requirements Analysis

The expected operation of the system is that a client sends a request to any server. That server then streams a file to the client over a TCP connection. If the server fails at any point during the stream, the failure should be detected, and another server starts communicating with the client to continue the download from where it was interrupted.

To focus on the download aspect and maintain the simple case of static file downloads addressed in [1] we make four simplifying assumptions:

1. All files that can be requested are unchanging and present at all servers.
2. All of the servers are initially healthy and able to communicate with each other.
3. The servers (collectively referred to as the server group) all have knowledge of each other's identities.
4. A client initially knows the identity of any alive server in the server group.

5. Design

The system consists of four components:

- the default behaviour of the client and server to download a file
- the heartbeat mechanism to monitor the server health
- the state synchronisation mechanism
- and the connection migration mechanism

5.1. Client-Server Interaction

To begin a download, the client initiates a TCP connection with a server. The client then transmits to the server the name of the file requested. The server then sends the total size of the file in bytes, and the client responds with a byte offset. The byte offset dictates how far into the file the server will begin transmitting, skipping that many bytes. Then the server sends the file from the byte offset until the end of the file closing the connection. The client determines if the transmission was complete by comparing the number of bytes received to the transmitted file size.

5.2. Health Monitoring

The servers detect their peers' liveliness by using a heartbeat mechanism. If a sufficient interval passes without receiving a heartbeat from a server, it is assumed that the server has failed. The heartbeat interval cannot be made lower than the round-trip-time between the two servers. A lower heartbeat interval increases the rate at which failure is detected. However, it also increases resource usage, increasing overheads [9]. Lowered heartbeat intervals may also cause over-eager declarations of server death under congested conditions.

Soft-state Synchronisation

The aim of the soft-state synchronisation mechanism is to ensure that for all clients currently communicating with a server, at least one other server is aware of their communication. This cannot always be satisfied as server failure before any synchronisation message can be sent prevents this record from being made and is an inherent limitation of the system design. However, we can attempt to minimise this possibility by ensuring that every server in the server group has an accurate understanding of the server groups current connections.

We must have this state synchronicity to ensure that all connections a server has can be resumed by another server in the server group. In our case, this is merely the client's identity (IP address and port number) since all other information is provided by the client upon connection with the new server (see section 5.4). This is the servers soft-state, and any additional state information can be lost without consequence.

We intended to modify the implementation of synchronisation used by Snoeren et al. [1] to reduce traffic load between servers. Snoeren et al. advocate for transmitting the entire list of connections a server has whenever a change occurs. A soft-state change occurs when a new client connects to the server or when a stream is completed and the connection to the client closed. Instead, we only transmit the change in the state, i.e. the connection that has been initiated or terminated.

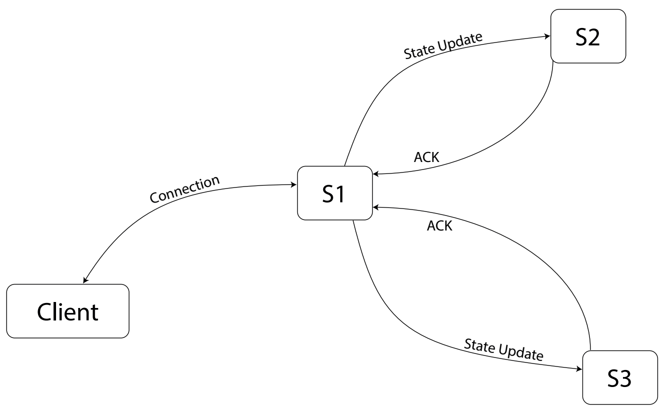


Figure 1. Diagram of State Synchronisation mechanism

All state change transmissions include the following information:

- The client IP address.
- The port the client is listening to for migration requests.
- If the connection is new or terminated.

Using this information, each server can update their local understanding of the server groups' state.

5.3. Connection Failover

When the heartbeat mechanism detects a server fault, it triggers a connection migration. The client initially opens a port that it can be contacted on to inform it that a server failure has occurred. When the client receives this message, it terminates its current connection and establishes a new connection with the server that contacted it.

Since the client tracks the number of bytes received so far, the client transmits that value as the byte offset when establishing the new connection. The stream begins from the next non-received byte in the file and the client continues to receive the file contents from the new server.

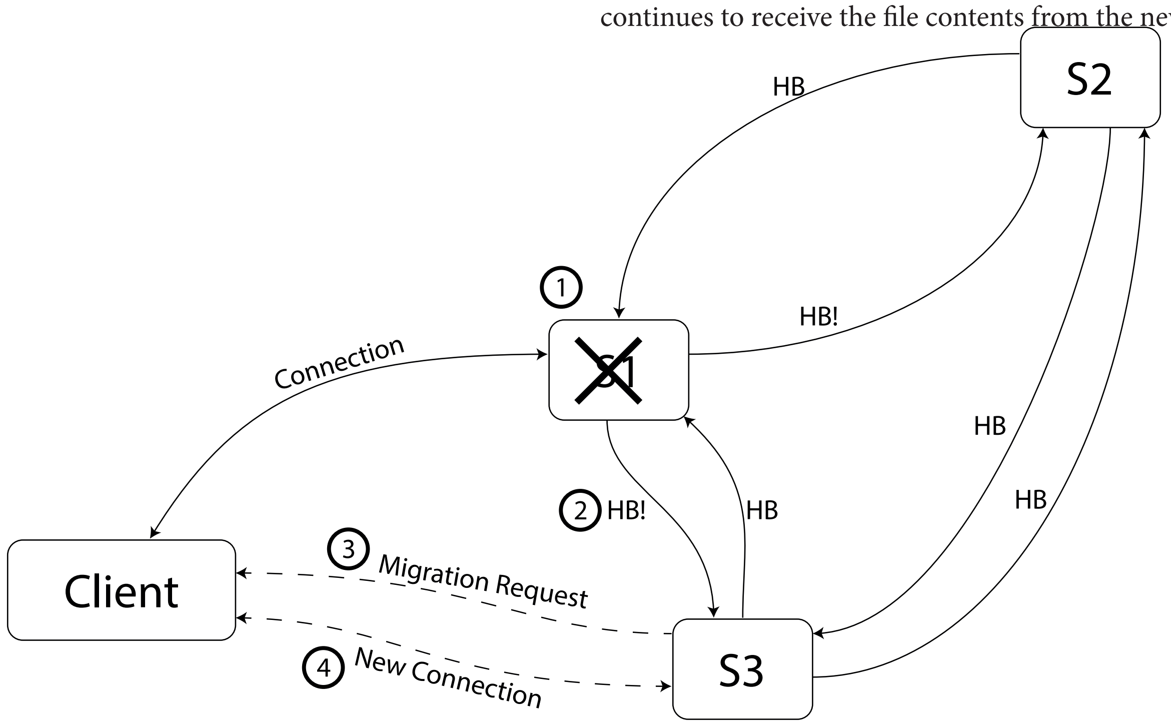


Figure 2. Connection Migration overview. 1. The server dies. 2. It does not sent heartbeat requests. 3. This triggers server 3 to send a migration request to the client. 4. The client initiates a new connection with server 3.

5.4. Model Robustness

The system, as described, should correctly handle failures where the server stops communicating with the client and at least one other server in the server group correctly. If only communication with the client fails, then the server group never prompts the client to reconnect, and the download never continues.

Another case that the model does not account for is when the client dies. Since the system intends to provide a highly-available service, all measures to improve reliability act on the server and the client-server connection, not the client itself. In most cases, a client failure completely prevents a transmission though it is feasible that a client could restart itself and request the remainder of the file by using the appropriate byte offset.

A final edge case of note is the expected behaviour when a server to server connection fails and a client is incorrectly informed that the connection has terminated. There are two options in this case: the first is to ignore the request, the second is to migrate regardless.

If the TCP connection between the client and server can close itself via time-out, the migration request can be ignored unless the server times out shortly afterwards (the request can be cached to store reconnection details). This prevents the slight overhead of terminating the existing connection and establishing the new one.

Alternatively, always following these requests require that the client is always listening to the migration request port, which can be achieved using non-blocking sockets or multi-threading the client-side application. This has the advantage that no-extra features or implementation are required for the TCP connection though additional implementation work and overhead are necessary for the client application.

6. Implementation

Here we provide an overview of additional factors that affected the design's implementation, some comments on project management and instructions on running the code.

6.1. Remote Collaboration

Owing to the pandemic, all work on this project had to be completed remotely. The initial meetings took place, at least weekly, over Microsoft Teams. These meetings aimed to investigate the topic, define the concrete issue to tackle, and gather information on implementation mechanisms.

The team was split up and self-assigned to tasks based on an online Kanban board, which we all organised. A remote Git repository was created, allowing team members to work on their assigned tasks on separate branches. We aimed to program in pairs using a Visual Studio Live Share session hosted at all times; however, this proved problematic since files sometimes would not save or even delete themselves entirely upon renaming them. Source control features were also not fully functional, occasionally unable to track any changes made.

As a result, the use of technology Live Share ended, and pairs instead used the screen share features of Discord and Microsoft Teams as they proved more reliable even if slightly less feature-rich.

6.2. Health Monitoring

A server monitors its peers' heartbeat across two threads, one for reading incoming heartbeats and a second for transmitting heartbeat packets — a typical multi-threading approach for network code. The heartbeat is implemented using TCP rather than the more standard UDP. The idea was that this would ensure that the heartbeat would get through if it was present so that fewer heartbeat failures were needed to trigger a migration request. In

```
Function Heartbeat(List of IPs, Port Number):

    Receiving Thread:
        Create a new socket instance
        Bind the socket to port number
        Listen for new incoming connections from the other servers
        Loop:
            Accept incoming request
            ➔ Create new connected socket + Get Sender IP
            Create new thread for communication over new socket +
            sender IP

    Sending Thread:
        Wait for other servers to start up
        For each IP in IP_List: //pairwise connections
            Create new socket
            Connect to server IP
        Loop:
            For each IP in IP_List:
                Send HEARTBEAT through Socket + IP
                Wait some time (dependent on heartbeat rate)
```

Figure 3. Pseudocode of heartbeat mechanism

practice, this significantly increased the complexity of the implementation and led to troubles with the multi-server case.

6.3. Soft-state Synchronisation

States are stored as linked lists of server data structs with each server data struct having its own linked list of connections. Like the heartbeat mechanism, implemen-

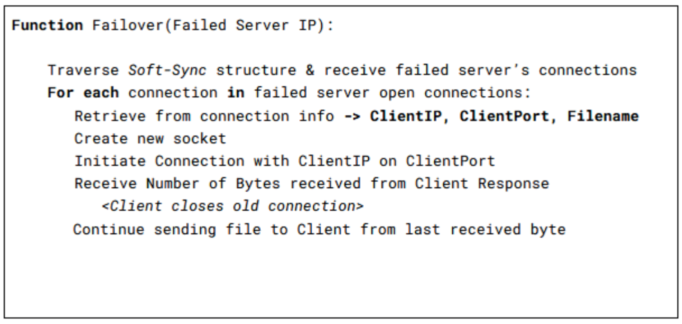
```
State Storage
server A -> conn1-conn2-…
   |
server B -> conn1-conn2-…
   |
server C -> conn1-conn2-…

Conn
    -   Filename
    -   Client IP
    -   Client port
```

Figure 4. Data structure for tracking server connections

tation was across two threads: one for listening and one for sending. The sending can be triggered from another thread and otherwise waits to be initiated.

6.4. Connection Failover

Implementing connection failure was relatively simple as much of the groundwork was pre-existing, and the additional complexities of multiple threads were not relevant. When a server is informed that a heartbeat has failed, it immediately navigates the linked list of states to send reconnection requests to all of the clients listed.



```
Function Failover(Failed Server IP):

    Traverse Soft-Sync structure & receive failed server's connections
    For each connection in failed server open connections:
       Retrieve from connection info -> ClientIP, ClientPort, Filename
       Create new socket
       Initiate Connection with ClientIP on ClientPort
       Receive Number of Bytes received from Client Response
          <Client closes old connection>
       Continue sending file to Client from last received byte
```

Figure 5. Server-side failover implementation

The client-side implementation did use another thread as we decided to use the always migrate option described in section 5.4.

6.5. Running the Code

The code runs using Docker with the intent to more easily replicate the conditions of the use-case in a lightweight manner. The sand-boxed nature of Docker also meant that testing the code could occur on any machine without concern of custom setups and modifications interfering.

In the root directory of the code is the docker-compose.yml on line 44, the final command-line argument passed to the execution of the client code is the name of the file to download. This can be any file in the server/ directory, and the downloaded copy will be written in the client/ directory.

To begin executing the two server, one client case run the command docker-compose up --build on a machine with Docker installed. In a separate terminal instance (or having backgrounded the initial call) run Docker kill server-1 to terminate the server the client is communicating with. It is important to note that the delay in the transfer was artificially implemented. This would aid with testing later due to combat the fast transfer times.

7. Testing

The use of Docker automates initialising the servers and client. A bash script is used to automatically manipulate Docker for our needs, and as a result, performs system testing. The docker-compose file already places any files the client receives in the client/ directory, allowing for easy comparison against the original server file. The script tests one client connecting to one server (server-1), with another backup server running in the background (server-2). The bash script currently kills server-1 mid-transfer. Ideally, we would kill the server based on when the transfer has started to ensure that failover occurs, but using a time-out was consistent enough.

Multiple runs of the script had the system yield the expected result: Server-2 took over the connection, allowing the client to receive the remainder of the file successfully. This was verified with the cmp command provided in Linux. Occasionally, though, Docker would fail to correctly run the containers and set up any virtual networking infrastructure. Such faults were rare, and since a timing issue regarding Docker causes this error, it was ignored.

To further check the failover system, testing was performed on a small video file (1.75 gigabytes in size), with server-1 killed thirty seconds into the transfer. The expected output of a correctly received file was seen within the client folder, again verified with cmp. Since our implementation only works for the two-server case and does not account for servers coming back online, system testing was limited to running a single failover request multiple times. This is opposed to running numerous failover requests within the same connection or download.

Tests were executed all on the same machine, which meant download times were not accurate. Routing issues and delays caused by intermediate relays were a non-factor, but there was insufficient access to physical geographically dispersed servers for more robust testing.

	Transfer time for file sizes			
	223MB mp4	12MB mov	49KB txt	6.6KB txt
Without failure	1228.8ms	69.051ms	1.767ms	1.291ms
With failure	144.3ms	124.53ms	4.940ms	2.117ms

8. Evaluation

TThe solution developed is functional and sufficiently meets the criteria for a subset of the cases desired. Implementation is correct for the two server case where there are at most a single server failure. Files of any type are correctly transmitted from the server to the client provided at most one server fails during the transmission. Two factors cause this. First, modifying the soft-state synchronisation protocol to only broadcast changes caused unexpected difficulties with server resurrection or restart. The second is that the heartbeat mechanism does not correctly recognise when a server that had ceased communicating is resurrected. A root cause of both of these issues is the use of TCP instead of UDP. The connectionless nature of UDP makes it well suited to the transient and one-off inter-server communication needed; the benefits we perceived with a TCP implementation were not realised.

Another issue with the implementation arose from the use of Docker as an execution mechanism. Whilst intended to simplify building and executing the code, this also made it challenging to test the N-server and multi-client case during development. The existing implementation was insufficient to handle those cases and was discovered too late to make the significant changes needed to rectify this. It would have been better to have the server and client code run on local machines and pass in operational parameters with command-line arguments.

The most critical shortcoming is the inability to correctly handle a server being restored, as the probability of more than one failure during transmission is negligible. Many existing implementations do not extend to the N-server case and provide high availability.

Outside of these areas, though, the underlying implementation proved sound, and with some refactoring work and reimplementation of the heartbeat and soft-state mechanisms, the protocol shows potential to have utility.

9. Future Work

Aside from the above changes necessary to complete the original scope, some additional features could be implemented. The first is to allow servers to update their content and inform clients that the download must be restarted as the file has updated. This would require modification of the initial connection setup and the soft-state synchronisation to include the file identity and the file's last modification date. When a connection is migrated, the server offering the new connection can inform the client that their previous download has been replaced and restart the file download to deliver the updated version.

References

[1] Snoeren, A., Andersen, D., and Balakrishnan, H. Fine-Grained Failover Using Connection Migration. In *USITS*, vol. 1. 2001.

[2] Alvisi, L., Bressoud, T., El-Khashab, A., Marzullo, K., and Zagorodnov, Z. Wrapping server-side tcp to mask connection failures. In *Proceedings of Infocom 2001*. 2001.

[3] Marwah, M., Mishra, S., and Fetzer, C. TCP Server Fault Tolerance Using Connection Migration to a Backup Server. In *Proceedings of IEEE Int. Conf. on Dependable Systems and Networks*, pp. 373–382. 2003.

[4] Vogels, W., Van Renesse, R., and Birman, K. Six misconceptions about reliable distributed computing. In *Proceedings of the 8th ACM SIGOPS European workshop on Support for composing distributed applications*, pp. 276–279. 1998.

[5] Fetzer, C, and Suri, N. Practical aspects of IP takeover mechanisms. In *The Ninth IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pp. 250–250. 2003.

[6] Snoeren, A., and Balakrishnan, H. An End-to-End Approach to Host Mobility. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pp. 155–166. 2000.

[7] Sultan, F., Srinivasan, K., Iyer, D., and Iftode, L. Migratory TCP: Connection Migration for Service Continuity in the Internet. In *Proceedings of the 22nd Internation Conference on Distributed Computing Systems (ICDCS'02)*, pp. 469–470. 2002.

[8] Fox, A., Gribble, S., Chawathe, Y., Brewer, E., and Gauthier, P. Cluster-Based Scalable Network Services. In *Proceedings of the sixteenth ACM symposium on Operating systems principles*, pp. 78–91. 1997.

[9] Aghdaie, N, and Tamir, Y. Fast transparent failover for reliable web service. In *International Conference on Parallel and Distributed Computing and Systems*, pp. 757–762. 2003.