Comparison of Request/Reply Protocols

Cloud and Distributed Computing: Practical Essay

1 Introduction

Request-Reply protocols are at the heart of every distributed system, as communication is a necessity. With the rise of the internet, one of the largest distributed systems, it has been shown that a good understanding of the protocols available, their applications, their limitations, and how they handle errors is critical to a modern-day developer. In this report, we compare and analyse three different protocols that have been studied throughout the module and within the labs. Starting off, we give an overview of each of the protocols and how they specified. From this we derive issues of each protocol, as well as its uses. Finally, we compare each protocol in terms of real-life coding, comparing ease-of-coding and other aspects within the context of developing a chat application. From each of these sections, we can then summarise each protocols positives and negatives, in terms of both theoretical and practical studies.

2 Protocol Overview

2.1 TCP/IP

TCP/IP are a set of protocols that are used to facilitate communication between processes over networks. The Internet Protocol (IP) is a protocol that handles sending fixed-size packets of data to/from an IP addresses on a network[3]. This is done within the Internet layer, as seen in Figure 1. Transmission Control Program (TCP) is a protocol within the Transport layer responsible for sending messages between processes. It aims to create a more reliable connection than IP alone. To this aim, TCP is connection-orientated, meaning that connections must be negotiated and explicitly started/stopped by both sender and receiver[4]. Messages are allowed to be arbitrarily long, splitting them into segments able to be sent via IP where needed. Furthermore, each segment is given a 'sequence number' which dictates in what order the segments should be recombined at the receiver's end, along with including a checksum. Once a segment has been received, an acknowledgement (ACK) is replied. If the sender does not receive an ACK for a particular segment, that segment will be retransmitted[5]. TCP also allows for control of the rate at which messages may be sent. This is done by including a 'window size' with each ACK, which defines the amount of data that is allowed to be sent before blocking until the next ACK[3].

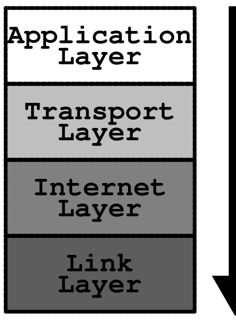


Figure 1: *The TCP/IP protocol stack; lower layers are closer to hardware, higher is more abstracted[1],[2].*

TCP/IP programming via a set of system calls, usually done through sockets[5]. Sockets are an interface used as the endpoint of process-to-process communication. For a server, a socket is created and 'binds' a process to a given port number. This socket then advertises its availability for a connection and blocks until a connection request is made. Then, communication can be done via simple `read()` and `write()` system calls. For a client, the socket does not need to be bound to a port explicitly, instead the socket tries to connect to a given IP address and port combination[4]. Once a connection has finished, the sockets must be closed. This process is visualized in Figure 2.

It should be noted that TCP can incur an overhead for small transactions due to needing to explicitly set up and close a connection before sending messages. TCP will declare a broken connection and shutdown if there is high packet loss, severe congestion on the network or if the network has failed entirely. Because of this, there is no guarantee for any message, including an ACK, to reach its

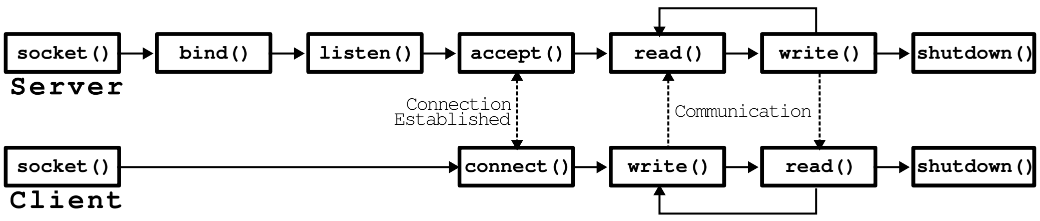


Figure 2: *Typical socket communication using system calls. Adapted from [4].*

destination, therefore TCP cannot be described as a fully reliable protocol[6]. Due to how the TCP checksum is implemented to prioritise speed rather than security, the checksum is quite weak in comparison to higher-level protocols and cannot be used to ensure integrity or validity of the message[7] and another protocol should be used.

2.2 HTTP

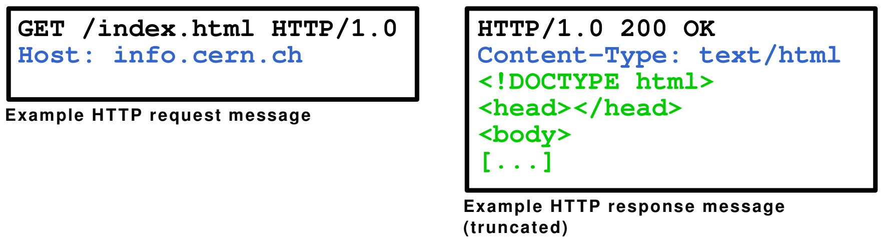
HTTP is a higher-level protocol usually implemented on top of TCP/IP within the Application layer. It is usually used to make requests to/from a web server and supports a fixed set of methods for communication[3]. The most common methods are[4]:

GET Request to return document to client.

HEAD Request to return header of document to client.

POST Provide data for a document.

HTTP only has two types of messages: request and response. Messages consist of 3 parts, first of which is the request line (for request) which specifies the method to use, what document to use the method on and the version of HTTP. Alternatively (for response), a status line is given, which specifies the status code, reason for status, and the HTTP version used. Next are the message headers, specifying information about what the client can handle or additional details from the server about the document. Finally, an optional message body which is typically used for a retrieved document contents[3]. Since messages may include additional data, the protocol allows for rudimentary authentication via credentials needing to be sent to the server and verified via the use of a challenge, such as a phrase that is hashed with a given password[3].


```
GET /index.html HTTP/1.0
Host: info.cern.ch
```
Example HTTP request message

```
HTTP/1.0 200 OK
Content-Type: text/html
<!DOCTYPE html>
<head></head>
<body>
[...]
```
Example HTTP response message (truncated)

Figure 3: *Example HTTP request and response messages. Each section of the message is highlighted.*

Sending messages is implemented using TCP usually. Because of how HTTP is designed, no attempt is made to recover from lost messages, and both client and server assume that the message has been sent[4]. HTTP supports two different methods of communication. One is persistent communication, where the TCP connection established stays open for more requests until explicitly closed by either client or server. The other method is non-persistent communication, where the connection is closed after one request-reply has been done. Persistent requests reduce latency due to not needing to re-establish a connection but is only applicable for connections that need to send lots of files and have bandwidth to spare[8]. Some clients allow the sending of multiple requests in a row to further decrease latency at the cost of more bandwidth usage[4].

2.3 Java RMI

Java RMI is an implementation of Remote Method Invocation (RMI) for Java-based programs. It aims to make programming for distributed systems similar to programming for monolithic systems, by hiding most of the detail in how the methods are transmitted and received. Java RMI, like HTTP, is implemented on top of TCP/IP. Since both processes are running on the Java virtual machine, data marshalling is not an issue[3]. In order to make remote methods visible, they must have an interface that extends the Remote class. This interface defines the return type and parameters the methods can take. Parameters may also pass-by-reference, allowing a remote method to access the object passed via RMI as well. This would be near impossible to implement in TCP/IP and HTTP. Due to this, classes may also be 'downloaded' to one another, implemented via the inbuilt object serialization methods.

In order for the process to find the interface(s) necessary, the RMI Registry must be running. The RMI Registry is a name service that allows the server to map an object that implements the Remote interface to a name, akin to how DNS maps URLs to IP addresses[9]. This allows the client to query the RMI Registry for an object with a given name, which then allows the client to receive the remote interface.

3 Programming Comparison

Throughout the labs, we have developed example programs for each of the three protocols in Java. In this section we compare each protocol's ease of coding, length of code needed, robustness and security of code. For certain points, the comparison will be put into the context of a chat application, for which the specification can be found at Appendix D.

3.1 Length of Code

As can be seen in Appendices A, B and C, each client program only sends one request, and each server only handles one request/one type of request. Below is a table listing line lengths for each package, generate by use of the `cloc` program.

Protocol	Client Length	Server Length	Total
TCP/IP	25	24	49
HTTP	25	20+29	74
Java RMI	18	25+7	50

Table 1: *Line length of each package, not including blank lines or comments.*

As can be seen from Table 1, each client package tends to stay similar in length no matter the protocol used. In contrast, the length of the server package tends to increase with higher-level protocols, potentially due to needing multiple classes to handle requests. It can be assumed that for increased functionality, line length would increase. For Java RMI, it can be assumed that line length would occur more within the server package, due to it containing both the interface definition and the implementation of each method needed. For TCP/IP and HTTP, it can be assumed that line length would occur roughly proportionally for both client and server due to the added overhead of constructing and/or decoding of different requests.

3.2 Complexity of Code

The IEEE Standard Computer Dictionary the term code complexity is defined as "The degree to which a system or component has a design or implementation that is difficult to understand and verify"[10]. From this definition, we can say that more complex code tends to consist of multiple objects, or tends to be based upon a more complex design. In terms of the design complexity of these protocols, I would say that Java RMI has a more complex design than HTTP and TCP/IP, mainly due to the idea of needing a RMI registry in addition to the processes running. However, implementation complexity varies depending on the program's needs. For a chat application needing to call methods from a simple API [1], it can be seen that Java RMI would potentially be a better option due to not needing to marshal the method calls into messages for transmission. HTTP, which consists of a simple design with a fixed set of methods and a stateless design, becomes much more complex due to the limitations of the simple design. Potential ways to subvert the fixed set of methods would be to include the actual methods called into the HTTP header or body instead, or to use the fixed set of methods as they are but to force the client and server to interpret them differently.

From these points, we can determine that in terms of design complexity, Java RMI is the most complex, then TCP/IP, with HTTP having the simplest design. In contrast for implementation complexity, HTTP would be the most complex, then TCP/IP, then Java RMI being the simplest.

3.3 Error Handling and Recovery

As described previously, TCP/IP includes a checksum in each message by default. However this checksum is quite weak in comparison to newer methods, and if a checksum error occurs, no attempt to correct the message is made, instead ignoring the packet and requesting a retransmission[5]. TCP/IP also does not distinguish between failures, so little can be done in the face of an error. HTTP, being built upon TCP/IP also does not distinguish between network errors and process failures, however due to its stateless design no guarantee is ever put on receiving a response and messages can be easily retransmitted. HTTP also includes status codes in a HTTP response, with well-defined codes for a myriad of errors along with a plain-text definition of the error in the message body. Java RMI has excellent error handling with a variety of Exceptions implemented for different failures[11], however error recovery for most errors is dependant on the programmer implementing it (if possible). Below lists a table showcasing it's effectiveness at handling errors and recovering from errors.

Protocol	Error Handling	Error Recovery
TCP/IP	1/5	2/5
HTTP	4/5	1/5
Java RMI	5/5	3/5

Table 2: *A table ranking each protocol for its error handling and recovery ability. A higher score is better.*

For a chat application specification we have designed, error handling is somewhat important, but not critical to its function. Ideally, the chat application would be able to recover via retransmission of data and to have verbose error handling if an error occurs. This would need to be implemented by the programmer if TCP/IP is used. In Appendix D.3 is a potential format for implementing error handling regardless of the underlying protocol.

3.4 Modularity

Modularity, for the chat application specifically, is the ability to increase the amount/types of messages passed between client and server with as little change to the code as possible. As described previously, HTTP is inherently poor at this due to the limitations of the fixed set of methods. Java RMI easily supports more functions and calls, as all that would need to be changed is to add another function to the interface and server implementation. However, for TCP/IP modularity can be done but is still limited by its design. Since TCP/IP is a lower-level protocol and only sends streams of bits[5], TCP has no idea of data types. Therefore if both the client and server needed to specify an additional data type, both the client and server would need to ensure they knew the representation of the new data type sent. However, for a chat application, this is less necessary as all messages can be marshalled to characters for ease of transmission, especially since Java already includes functions for converting Strings to a given data type[12].

[1]Refer to Appendix D for a list of the methods needed for the chat application.

3.5 Security

Security is a concern for a chat application, as security issues could allow users to spoof other users, or that the server could expose client's IP addresses in extreme cases. TCP/IP inherently does not implement much security due to the protocol being lower-level than other protocols. TCP/IP messages are not encrypted in any way and would require a higher-level protocol to offer that functionality[5] HTTP by default does not offer much security due to being implemented on top of TCP/IP. However, this can be rectified by using HTTPS, which is instead implemented on top of TLS and requires signed certificates for communication[13]. Likewise, Java RMI can be configured to block any traffic not from another Java RMI process, and both machines can be secured with TLS. However, since Java RMI and HTTP both run atop TCP/IP, they are not initially secure.

4 Summary

In summary, we have looked at how all three protocols work, their benefits and negatives for a given chat application. From the analysis shown above, each protocol excels at a different section. For a chat application specializing in secure communication, HTTP or RMI over TLS/SSL would be a good option to mitigate security flaws. For a chat application that is expected to increase in complexity or functionality, Java RMI would be a good option due to its use of interfaces and ease of adding remote methods. For a very simple chat application, like the one specified in Appendix D, TCP/IP would be best due to the small length of code needed to set up and the lack of design complexity. Overall, from this analysis, modern-day distributed system designers should have a good understanding of the implications of using a particular protocol and can extend what they have learnt into other real-life applications.

References

[1] R. T. Braden, *Requirements for Internet Hosts - Communication Layers*, RFC 1122, Oct. 1989. DOI: 10.17487/RFC1122. [Online]. Available: https://www.rfc-editor.org/info/rfc1122.

[2] R. T. Braden, *Requirements for Internet Hosts - Application and Support*, RFC 1123, Oct. 1989. DOI: 10.17487/RFC1123. [Online]. Available: https://www.rfc-editor.org/info/rfc1123.

[3] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. Pearson Education Ltd., 2012.

[4] A. Tanenbaum and M. van Steen, *Distributed Systems: Principles and Paradigms*. Pearson Education Ltd., 2007.

[5] *Transmission Control Protocol*, RFC 793, Sep. 1981. DOI: 10.17487/RFC0793. [Online]. Available: https://www.rfc-editor.org/info/rfc793.

[6] S. Porter, *Communications 1/2*, Lecture, 2023.

[7] J. Stone, M. Greenwald, C. Partridge, and J. Hughes, "Performance of checksums and crcs over real data," *IEEE/ACM Transactions on Networking*, vol. 6, no. 5, pp. 529–543, 1998. DOI: 10.1109/90.731187.

[8] S. Porter, *Communications 2/2*, Lecture, 2023.

[9] Oracle. "Java platform se tools reference: Rmiregistry." (2023), [Online]. Available: https://docs.oracle.com/javase/8/docs/technotes/tools/unix/rmiregistry.html (visited on 01/07/2024).

[10] "Ieee standard computer dictionary: A compilation of ieee standard computer glossaries," *IEEE Std 610*, pp. 1–217, 1991. DOI: 10.1109/IEEESTD.1991.106963.

[11] Oracle. "Java remote method invocation specification appendix a: Exceptions." (2023), [Online]. Available: https://docs.oracle.com/en/java/javase/21/docs/specs/rmi/exceptions.html (visited on 01/07/2024).

[12] Oracle. “Java language specification: Chapter 5.” (2011), [Online]. Available: `https://docs.oracle.com/javase/specs/jls/se7/html/jls-5.html` (visited on 01/07/2024).

[13] R. T. Fielding, M. Nottingham, and J. Reschke, *HTTP Semantics*, RFC 9110, Jun. 2022. DOI: `10.17487/RFC9110`. [Online]. Available: `https://www.rfc-editor.org/info/rfc9110`.

A TCP/IP Client/Server Program

Below lists the simplest TCP/IP client and TCP/IP server. The client sends one GET request and awaits a response before closing. The server awaits one message, responds with a simple reply message before closing.

A.1 client/Client.java

```java
package client;

import java.net.*;
import java.io.*;

/**
 * A simple single-threaded TCP/IP client. One request is sent and the response
 * is sent to stdout before closing.
 */
public class Client
{
    private final static String DEFAULT_HOST = "localhost";
    private final static int DEFAULT_PORT = 8000;

    public static void main( String[] args )
    {
        try {
            // create socket and accept server
            Socket sock = new Socket( DEFAULT_HOST, DEFAULT_PORT );

            // create io streams
            BufferedReader in = new BufferedReader( new InputStreamReader(
                                                    sock.getInputStream() ) );
            PrintWriter out = new PrintWriter( sock.getOutputStream(), true );

            // request GET and block until response is given, then print
            out.println( "GET" );
            String response = in.readLine();
            System.out.println( "Response was: \"" + response + "\"" );

            // close io streams and socket
            in.close();
            out.close();
            sock.close();
        } catch ( Exception e )
        {
            e.printStackTrace();
        }
    }
}
```

A.2 server/Server.java

```
package server;

import java.net.*;
import java.io.*;

/**
 * A simple single-threaded TCP/IP server that responds to one message before
 * closing.
 */
public class Server
{
    private final static int DEFAULT_PORT = 8000;

    public static void main( String[] args )
    {
        try {
            //start server socket and wait for client to accept
            ServerSocket s_sock = new ServerSocket( DEFAULT_PORT );
            Socket c_sock = s_sock.accept();

            //create io streams
            BufferedReader in = new BufferedReader( new InputStreamReader(
                                    c_sock.getInputStream() ) );
            PrintWriter out = new PrintWriter( c_sock.getOutputStream(), true );

            //block until request is received
            String request = in.readLine();

            //send response message
            out.println( "You sent request: \"" + request + "\"" );

            //close io streams and sockets
            in.close();
            out.close();

            c_sock.close();
            s_sock.close();
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }
}
```

B HTTP Client/Server Program

Below lists a simple HTTP client and server. The server only handles GET requests. The client only sends one request, prints the response and closes.

B.1 client/Client.java

```java
package client;

import java.net.URI;
import java.net.http.*;
import java.net.http.HttpResponse.*;

public class Client
{
    private static final String DEFAULT_HOST = "localhost";
    private static final int    DEFAULT_PORT = 8000;

    public static void main( String[] args )
    {
        try {
            // construct URI of server , i.e. http://localhost:8000
            URI uri = new URI( "http", null, DEFAULT_HOST, DEFAULT_PORT, null,
                               null, null );
            // create httpclient to handle requests
            HttpClient client = HttpClient.newHttpClient();

            String requestMsg = "Custom message!";

            // construct a HTTP request, defaults to GET request
            // adds a header with a custom message requestMsg
            HttpRequest request = HttpRequest.newBuilder().uri(uri)
                                 .headers("message", requestMsg).build();
            //sends message and blocks until response is given
            HttpResponse<String> response = client.send( request,
                                                    BodyHandlers.ofString() );
            //prints response
            System.out.println( "Response was: \"" + response.body() + "\"" );
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }
}
```

B.2 server/Server.java

```
package server;

import server.RootHandler;

import java.net.*;
import com.sun.net.httpserver.*;

public class Server
{
    private final static int DEFAULT_PORT = 8000;

    public static void main( String[] args )
    {
        try {
            // create http server on port 8000
            HttpServer srv = HttpServer.create( new InetSocketAddress(
                                                DEFAULT_PORT ), 0 );

            // give class that handles requests to root directory
            srv.createContext( "/", new RootHandler() );
            srv.setExecutor( null ); // use default executor
            // start server
            srv.start();
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }
}
```

B.3 server/RootHandler.java

```java
package server;

import com.sun.net.httpserver.*;
import java.io.*;

public class RootHandler implements HttpHandler
{
    private void handleGetRequest( HttpExchange exchange ) throws IOException
    {
        // strip custom message from header, then send response message
        // with header
        Headers headers = exchange.getRequestHeaders();
        String message = headers.get("message").get(0);

        String response = "You sent message: " + message;
        exchange.sendResponseHeaders( 200, response.length() );

        OutputStream out = exchange.getResponseBody();
        out.write( response.getBytes() );
        out.close();
    }

    @Override
    public void handle( HttpExchange exchange ) throws IOException
    {
        // handle requests sent to root directory
        // only deal with GET requests
        String method = exchange.getRequestMethod();
        switch ( method )
        {
            case "GET":
                handleGetRequest( exchange );
                break;
            default:
                break;
        }
    }
}
```

C Java RMI Client/Server Program

Below lists a simple Java RMI client/server example. The client only calls one remote method, then closes. The server only implements one method to be called.

C.1 client/Client.java

```java
package client;

import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;

import server.TestInterface;

public class Client
{
    public static void main( String[] args )
    {
        try {
            // find registry running on localhost
            Registry reg = LocateRegistry.getRegistry( "localhost" );
            // get stub for interface
            TestInterface test_stub = (TestInterface) reg.lookup( "Test" );

            // call remote method
            String response = test_stub.testMethod();
            System.out.println( "Response was: \"" + response + "\"" );
        } catch ( Exception e ) {
            e.printStackTrace();
        }
    }
}
```

C.2 server/Server.java

```java
package server;

import java.rmi.registry.Registry;
import java.rmi.registry.LocateRegistry;
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;

import server.TestInterface;

public class Server implements TestInterface
{
    // remote method implementation
    public String testMethod()
    {
        return "Called remote method!";
    }

    public static void main( String args[] )
    {
        try {
            Server srv = new Server();
            // obtain stub of TestInterface object (Server)
            TestInterface test_stub = (TestInterface) UnicastRemoteObject
                                                     .exportObject(srv, 0);

            // find registry running on localhost
            Registry registry = LocateRegistry.getRegistry();
            // binds stub to given name
            registry.bind("Test", test_stub);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

C.3 server/TestInterface.java

```java
package server;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface TestInterface extends Remote
{
    String testMethod() throws RemoteException;
}
```

D Chat Program Specification

The chat program would aim to implement a chatroom program, where multiple clients could connect to a central chatroom and communicate to each other. No client-client communication would be implemented. The program aims for every request to be sent by the client for ease of programming, therefore clients would be responsible for timely fetching of new messages. The methods (abstracted from the underlying protocol) to be implemented would be as such:

join(username, host, port) Explicitly sets up connection if needed by underlying protocol. Asks to join server at 'host:port' with nickname 'username'. Should return 0.

leave() Notify the server that the client has disconnected. Explicitly closes connection if needed by underlying protocol. Should return 0.

send_message(message) Sends a message to the chatroom. Username and timestamp should be appended implicitly before sending message by underlying protocol. Should return 0.

get_backlog_amount(timestamp) Queries the server on how many messages have been sent since the given timestamp. Should return an integer denoting the amount.

get_messages(amount) Returns an array of Message objects of size *amount* or fewer. Since no integer is returned, instead treat an empty array as an error.

Upon any error, a corresponding error code should be returned instead, taking the form of a negative integer instead.

D.1 Server

The server would be responsible for correct ordering of messages (dubbed the chatlog) depending on the timestamp received within the `send_message()` request. Each new client that sends a `join()` request should be given some form of ClientHandler class and thread that facilitates responding to that client's requests. Since multiple threads could try to modify and/or read the chatlog at one time, each read/write should be queued for the main thread to respond, so that the server remains thread safe. Upon receiving a `leave()` request, the ClientHandler should close itself and any necessary resources. In lieu of an explicit `leave()` request, the ClientHandler should close itself if no request has been received in 5 minutes.

D.2 Client

Since the client is responsible for both sending messages and fetching new messages, the client should be implemented on multiple threads, one for dealing with the underlying protocol, where method calls would be queued to ensure no requests are sent while waiting for a response, another thread for dealing with checking backlog, fetching messages and displaying messages to the user, and a final thread for dealing with user input and sending messages. Potentially, this could be cut down to a single thread with efficient use of interrupts.

D.3 Message Format

Messages would be stored with these fields and data types. This implementation is inspired by how HTTP implements its replys and responses. For transmission in TCP/IP and HTTP, this Message object would need to be converted into an intermediary format.

```
PUBLIC CLASS Messages:
    Int  status_code
    Long timestamp
    String username
    String method
    String message_body
```

For `status_code`, the code is 0 for a successful operation and negative for an error. This field is only populated on server responses.
For `timestamp`, this is the timestamp of the message sent by the client or the message retrieved by the server.
For `username`, this is the username of the message sent by the client or the message retrieved by the server.
For `method`, this is the method called by the client. This should be copied to the message sent by the server's response.
For `message_body`, this is either the body of the message sent/retrieved, or a plaintext description of the error if the status code is negative. It may also be used to store the amount of messages backlogged when calling `get_backlog_amount()`.