FSSP (Fail-Safe-Simple Protocol)

Reliable and Simple Protocol For Humans.

Introduction

FSSP provides reliable communication over unreliable UDP protocol and also directly works with python objects for simplicity. This class provides API (application program interface) for the underlying reliable application layer protocol. An object of this class should be instantiated to set up communication with a peer. This protocol requires python3 to run and doesn't have any other dependency. All standard python packages are used to build this protocol.

It uses sequence numbers to number packets to keep a track of them, acknowledgement to confirm that packet reaches its destination and retransmission for packets who failed to reach its destination or its acknowledgement lost somewhere in the transmission.

Blocking and Non-Blocking

FSSP provides the application with two modes of communication i.e. blocking and non-blocking. Blocking means the method will stop the execution until the packet is sent or put in a reliable buffer and non-blocking is if the buffer is full or packet cannot be sent it will return back with an error message.

Application Program Interface (API) of FSSP

FSSP provides many methods for the application to interact with the underlying protocol, following is a list of methods, their parameters, their return value and functionality.

FSSP(*interface string, port integer*): Instantiate protocol's object. Here interface, port is the point of binding in the host system. The protocol will listen to packets at this interface and port.

connect*(interface string, port integer*): Store the address of the host to which packets will be sent. Returns None.

listen(): Start listening for packets (data or acknowledgement) on (interface, port) provided during instantiation. Returns None.

recv(*blocking=True boolean*): Return the next data packet in the sequence to application. By default this method is blocking, i.e. it will stall the execution of the overlying application until it returns something. Though users can pass '*blocking=False*' and then the recv will just check the buffer, if there is any data it will return data otherwise it will return *'False'* and return the execution back to the application.

Note: Application will receive data in the same sequence / order as it was transmitted by the peer no matter at what sequence they arrive at the host.

send(*data Hashable, blocking=True boolean*): User can send any hashable python object and FSSP will handle its serialization and transmission. Simple isn't it!

By default send is blocking, i.e. it will wait until the packet is pushed into sent buffer, though user can set it to non-blocking *'blocking=False'* and it will see if the buffer has space, it will push the packet into it otherwise it will return *'False'* and return the execution back to application.

Hashable: Any python object whose hash value does not change during its lifetime is hashable. Hashable objects which compare equal should have the same hash value. Examples of hashable classes in python are string, integer, float etc or derived data types like tuple.
User can even define a custom class which is hashable (__hash__() dunder method) and its object could very easily be sent over by the protocol, the user doesn't have to care about binary, encoding, decoding etc **(that's why it's for humans!)**

To check if an object is hashable or not:

```
from collections.abc import Hashable

if (isinstance(<your-object>, Hashable)):
      print("Object is hashable")
```

Best thing about this protocol is it's simplicity of working directly with python objects. And FSSP achieves this using the powerful tool bundled in the python standard library called **pickle,** which can serialize any python object into byte stream and vice-versa.

Consistency: FSSP maintains the consistency of data by providing an additional header called "hash" which is actually the md5 hash of the data sent by the overlying application. So when a peer receives the data it can calculate it's md5 hash and compare to check if any data corruption has occurred or not.

Protocol Configuration

PACKET_SIZE: It is the maximum size of FSSP packet (not user's data packet) which can be transmitted over by the protocol. It should be less than the MTU of the network, because intermediate routers are not allowed to break packets into smaller chunks.

WINDOW_SIZE: It is the maximum number of unacknowledged packets which can remain in the buffer. If it is full, more packets sent from the application are rejected. Similarly it's also the maximum size of the received buffer, if it's full more packets from peer are rejected.

TIMEOUT: It is the amount of time FSSP waits for the arrival of ACK of an already sent packet. If ack doesn't arrive until TIMEOUT seconds, the packet is retransmitted.

Selective Repeat ARQ

FSSP implements selective repeat for automatic repeat of requests (ARQ) i.e. it only retransmits packets which aren't ACKed yet and keeps all the ACKed packets (in window size limit) in the received buffer which greatly improves the performance at the cost of memory usage.

Flow of control: When a user sends data using 'send' method, data is packed in a dict with additional headers like hash value, sequence number and type of packet (acknowledgement or data) in a dictionary. The bundled packet is then sent to peer using UDP socket and also pushed to the sent buffer where it will wait till acknowledged or retransmitted if it is found to be waiting for more than TIMEOUT seconds.

1) **Listening thread**: This thread listens for incoming packets from peer, if the type of packet is ACK, it removes the respective data packet from the sent buffer. If it's a data packet, it is pushed to the recv buffer, from where it will be sent to application sequentially.

2) **Retransmission thread:** This thread checks the sent buffer every TIMEOUT seconds and all the packets which are timed out are retransmitted.

Thread Safety

The protocol implements mutex locks for all of its data structures and therefore the protocol is thread safe and the application can use it using multiple threads and it will work without any race conditions or inconsistent data.

Packet Structure

FSSP uses a python dictionary to bundle up it's packet. There are two different types of packets, one for data transmission and another of acknowledgement. Finally the complete packet is dumped as a binary stream using **pickle** and sent via socket to peer.

```
{ # Data packet
   "seq": sequence_number <integer>,
   "data": user_data <hashable object>,
   "hash": md5_hash_of_user_data,
   "type": "DATA"
}

{ # ACK packet
   "seq": sequence_number <integer>,
   "ack_seq": seq_number_of_acked_packet <integer>,
   "type": "ACK"
}
```