**🔄 THE EVENT LOOP in XRootD**

**A high-level overview 🌟**

*[Clarifications marked with blue]* 😉

The event loop is a mechanism for getting feedback from the kernel [I assume that you are referring to a TCP kernel here. If so, there is a socket on the client and also a socket on the server, right?] whether

Feedback from linux kernel

- there is space in the TCP output buffer for writing

- there are data in the TCP receive buffer for reading

[size of the two buffers is established on the client-server handshake, right? And moreover, they can differ in size?]

The TCP buffer size is a system setting.

now, in the application there are two quests:

1) the queue of requests the client wants to issue to the server, whenever there is a write event we take a request from the queue and write it to the socket, note: the TCP buffer might not be big enough to accommodate the whole request, so it might be several write events will be needed to write a single request

1.1) every request is accompanied by a message handler, after the request is fully written to the socket, the message handler is being moved to the 2nd queue for the incoming responses. [so, one can imagine the response queue as being composed of a stack with pairs of Response-id/Message\_Handler-id, and with each successful request sending to the server, the Message\_Handler moves into the response stack?]

That’s correct

2) whenever the event loop yields a reading event it means we can read something from the socket, so we read out a server response, again it might be one will need several read events to read a full server response. [we – as the XRootD client – know when to read something from the socket using the epoll syscall on that file descriptor which was opened once the corresponding request was sent to the server, right?]

Mostly yes, just a comment the socket file descriptor is not necessary created when the request is being sent, we open the connection and then keep it open for reuse

2.1) once we read a full server response, we look in the queue for incoming responses for a matching message handler:

* once we have the handler, after parsing the response, we can call the user callback [the callback is more or less a function that runs whenever we get its corresponding response from the server, right?] yes :-)
* mind that we are executing the callback in a thread pool [not sure what the implications of the thread pool are: a limited/fixed number of execution workers to process the callbacks and responses from the server? Is that it? 🤔] yes :-)
* the syscall that is used for monitoring file descriptors is epoll [Is epoll “watching” the file descriptor that is started once we initiate the TCP connection to the server? Or with each request that we create to the server, a new FD starts. In the second case, that would imply a “Stack” of descriptors which the event-loop is monitoring] we have one connection per server, that said the epoll is ‘watching’ multiple file descriptors because the client might connect to multiple servers
* [from my understanding, a file descriptor is started/created when creating an interface between the client-space and the kernel-space - that is the TCP kernel which allows a connection between client and server] well, with the connect syscall you ask the kernel to establish a tcp connection, the file descriptor you are getting from that call if your interface for interacting with that physical connection

**Event-loop stages:**

The event loop generates ‘ready to read’, ‘ready to write’, ‘read timeout’ and ‘write timeout’ events. Accordingly, to those events, the client is sending requests and receiving responses, and handling timeouts. For connections using TLS, it is more complicated, but I think this is irrelevant for the paper.

Tentative for an event-loop diagram

Diagram

Description automatically generated