**Term Paper #3 – Final Network Protocol Design**

**Ankush Israney, Anthony Emma, Edwin Dauber, Francis Obiagwu**

**Service Description**

With the focus on cloud based computing in the past decade, several online services arose that allowed for users to share and edit documents in web browsers with other users seamlessly. These systems generally used simple revision control systems so that when a user made changes to the document, a new version of the document is created server side. This works well in many cases, but fails to be sufficient when multiple users are attempting to edit version X simultaneously. Thus a need was born for a new category of editors termed “Real-Time Collaborative Editors” (abbrev RTCE). With these editors a user can see other user’s updates being made at the same time allowing for improved collaboration and coordination amongst simultaneously users editing a document.

To coordinate changes we have envisioned a protocol for RTCE applications being built on a client/server model, with the server functioning to authorize, resolve, and commit updates to ensure data integrity within the document. We selected a client/server model over a peer-to-peer model because we believe the key difficulty in an RTCE will be maintaining the integrity of the document given an influx of simultaneous updates, and we feel that a server application provides a central well known and well trusted keeper of that document.

Our RTCE protocol effectively captures the types of basic control operations and interactions between the client and the server. At a high level we have identified some basic services our protocol supports to the application:

* User Authentication, Credentials, and Security handshaking
* Locking service for requests from the client to the server to lock a specified portion of the document. If the request is permitted the server will supply a token back to the client to use for the edit.
* Document commit service from a client to commit a set of changes to the document, which will be allowed if the client’s token provided with those changes matches the expected token (i.e., the token provided by the server granting access)
* Document update service which pushes document data from the server to the client
* Request to reclaim/release a locked portion of the document from another user.
* Logoff and Shutdown

We envision our protocol being built onto the transport layer protocol of TCP to provide a high level of reliability for the commands being issues and given the size of most messages is generally small.

Port number 50000 will be used for the TCP messages. As will be discussed below the protocol inherently supports multiple sessions, thus one port is sufficient to collect the data from multiple sessions and likewise multiple documents. In the future to address possible scalability issues on the server using just one port, we envision details of the port to be used for client/server communication could be located inside the handshake itself, allowing a server which is supporting hundreds of clients to deflect sessions to different ports instead of one massively congested port on the server.

Lastly, focus was given on byte alignment to ensure fields align on 32-bit boundaries where appropriate. We understand that most operating systems and chipsets want 32-bit integers (for example) to align at a byte address divisible by 4 at a minimum.

**Message Definitions**

**Header PDU**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Representation** | **Byte Location** | **Description** |
| **Request** | 8 8 bit ASCII characters | 0..7 | Indication of the type of message (ex. "S\_LIST ") |
| **Session** | 64 bit unsigned integer | 8..15 | Session Identifier for a specific server/client communication |
| **Timestamp** | 64 bit unsigned integer | 16..23 | The time that this message buffer was updated by the application |
| **Checksum** | 32 bit unsigned integer | 24..27 | CRC Checksum of the payload of the message |
| **Reserved 1** | 32 bit unsigned integer | 28..31 | Reserved for Growth |
| **Reserved 2** | 32 bit unsigned integer | 32..35 | Reserved for Growth |
| **Reserved 3** | 32 bit unsigned integer | 36..39 | Reserved for Growth |

The header PDU is the first portion of every message exchanged in the protocol and is primarily used to determine the interpretation of the remaining contents in the message (subsequently referred to as the “payload”). This is done via the **Request** field which is an 8 ASCII character field that contain the request types, for example:

CUAUTH CONNECT S\_LIST S\_DATA

S\_TREQST S\_TRESPN S\_DENIED S\_DONE

S\_COMMIT ABORT ECHO BLOCK

LOGOFF LACK CACK S\_REVOKE

The remainder of the Request field should be blank filled (ASCII character 0x20 space).

The **Session** field is agreed by both the client and the server and provides context to the protocol of the agreements established during authentication and is discussed in the sections on CUAUTH and CONNECT. The Session field provides an inherent level of trust in communication and also facilitates the ability for a client and/or server to support multiple documents and connections simultaneously. The application layer will choose to handle an unexpected session field, with the two likely options being to respond back with a failure code, or not process the message.

The **Timestamp** field is used to indicate when the message contents were updated by the application layer. This provides debug and logging capabilities, but also extends to issues we envision when multiple clients attempt to interact with a similar portion of the document. The application layer may simply choose to operate as a FIFO with client requests, but we believe an alternate and fair application implementation may use the timestamp of the message to allow a late arriving message to be accepted over an earlier arriving message so as not to consistently penalize users on a slower less reliable connection.

The **Checksum** field is a basic checksum of the contents in the non-header portion of the message. This is meant to filter out improper application layer behavior that may be either providing garbage or invalid data in a portion of the message.

Three 32-bit fields were provided for **Growth**, and provides extensibility. The selection of three was generally arbitrary, but only a single 32-bit growth field did not seem sufficient to handle future versions of the protocol.

Finally, the version number was purposely not embedded into the header as it would be wasteful to exchange on every individual interaction. Thus versioning was moved into the authentication and session establishment processing.

**Initial Handshake PDU**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| “CUAUTH” PDU |  |  |  | |
| Field | Representation | Byte Location | Description |
| Version | 4 bytes | 40..43 | major, minor, sub, ext. |
| username | 20 character ASCII | 44..63 | username |
| authentication | 16 character ASCII | 64..79 | password or other authentication |
| document owner | 20 character ASCII | 80..99 | username of document owner |
| document title | 20 character ASCII | 100..119 | document title |
| num encrypt options | 32 bit unsigned integer | 120..123 | number of presented options n |
| encrypt option list | 8 character ASCII | 124..(123+8n) | encryption options |
| num other options | 32 bit unsigned integer | (123+8n)..(127+8n) | number of presented options m |
| other option list | 8 character ASCII | (128+8n)..(127+8n+8m) | other options |
|  |  |  |  | |
| “CONNECT” PDU |  |  |  | |
| Field | Representation | Byte Location | Description |
| Version | 4 bytes | 40..43 | major, minor, sub, ext. |
| server authentication | 20 character ASCII | 44..63 | server id key |
| encrypt option | 8 character ASCII | 64..71 | chosen encryption option |
| num other options | 32 bit unsigned integer | 72..75 | number of selected options m |
| other option list | 8 character ASCII | 76..(75+8m) | selected options |
| num shared secrets | 32 bit unsigned integer | (76+8m)..(79+8m) | number of shared secrets n |
| secret list | 16 bytes | (80+8m)..(79+8m+16n) | shared secrets |

|  |  |  |  |
| --- | --- | --- | --- |
| “CACK” PDU |  |  |  |
| Field | **Representation** | **Byte Location** | **Description** |
| none | none | none | none |

During the process of the initial handshake, the PDU is used uniquely. The first message sent by the client is the CUAUTH request, and the first message sent by the server is the CONNECT message. The CUAUTH message is unique with respect to the header as it is the only message for which the session number is 0. The session number in the header of the CONNECT response is the session number for the entire session. Otherwise, these two messages use the header in the same way as every other message.

The CUAUTH message body begins with the version number of the client. This version number is a 4-byte value. The first byte is the major version, the second byte is the minor version, the third byte is the sub-version, and the fourth byte can be used to identify certain significant extensions. The CONNECT message body also starts with a version number. This version number is either the same as the version number of the client if the server is the same version or newer, or is the version number of the server if it is an older version. If it is an older version, it is a matter of the differences between versions as to whether the connection should be continued with that older version or if the client should attempt to reconnect with the older version.

The next fields of the CUAUTH message are strings of standard 8-bit ASCII characters, with the NULL character used to fill remaining spaces if the original string is too short. The first such field is the username, which is 20 characters long. The second field is the password or other authenticating information, which is 16 characters long. The third and fourth fields are the username of the owner or creator of the document to open and the title of the document respectively, and each are 20 characters long. Server applications should be designed such that this is sufficient to uniquely identify any document. In contrast, the CONNECT message only has one 20-character corresponding identifier which is a server key, used to authenticate the server to the client. These lengths are within the standard range, and make for easily delineable messages by multiples of 8.

The CUAUTH message then provides the number of ranked encryption preferences as a 32-bit unsigned integer, which can be 0. Each of these preferences is then enumerated as an 8 character ASCII string. It then repeats the format for generic option requests. The CONNECT message, after the server identifier, provides the 8 character ASCII string encryption selection (which will be NONE if the length of encryption preferences was 0 or none of the selected options was available). It then provides a list of agreed upon options from the client’s request in the same format as the CUAUTH message. The final part of the CONNECT message is 32-bit unsigned integer identifying the number of shared secrets and then the list of them, with each shared secret as a 16-byte (128-bit) value. This value was chosen because most recommended security key lengths which are longer than 128 bits are even multiples of 128.

The CACK message is where the client acknowledges receipt of CONNECT request from the server during the initial connection establishment process. It is just a blank message for controlling purposes as LACK.

**Section List PDU Request = “S\_LIST”**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Representation** | **Byte Location** | **Description** |
| N | 32 bit unsigned integer | 40..43 | The number of elements in this message, N >= 1 |
| Section ID [1] | 32 bit unsigned integer | 44..47 |  |
| Section Start ID [2] | 32 bit unsigned integer | 56..59 |  |
| … | … | … |  |
| Section Start ID [N] | 32 bit unsigned integer | 40+N\*4 .. 43+N\*4 |  |

In the case of an update of the document from the Server to the Client, or initial load of the document from the Server the Section List message is transmitted to convey the order that sections will appear in the document (from top of the document to bottom). Each section of the document is given a unique Section ID, and the **Section List** message is a collection of N number of Section IDs.

**Section Data PDU Request = “S\_DATA”**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Representation** | **Byte Location** | **Description** |
| Section ID | 32 bit unsigned integer | 40..43 | The section ID |
| Length | 32 bit unsigned integer | 44..47 | Length of the data field |
| Data | General | 48..<END> | Data for this Section ID, defined by application |

After receiving the list of Section IDs in the Section List message the client will next receive a series of data messages containing the contents of the individual sections of the document.

The **Data** field itself is purposely left General as a section of the document could be ASCII, Image, XML, or any number of formats established at the application level. (Note: for our coding demonstration/implementation we are using just ASCII text)

**Request = “S\_TREQST”**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Representation** | **Byte Location** | **Description** |
| Section ID | 32-bit unsigned integer | 40..43 | The individual Section ID |
| Length | 32-bit unsigned integer | 44..47 | Length of Section request from start (Section ID) if predetermined by application. Length might be useful by end section- paragraph. |
| V bit | 1 bit | 48(bit 0) | Length is zero if V bit = 1 (Variable section length from start up till next section.) |
| U bit | 2 bits | 48(bit 1+2) | Urgent Request (Combination of urgency+priority of user used for determining response for instance, currently editing user from previous section would have urgency bit high) |
| R1-R29 | 29 bits | 48(bit 3)..51 | Reserved Additional bits left for extensibility of options in the PDU (left 0 in basic version) |

Token exchange is the heart of the RTCE protocol in terms of collaboration to edit. It is the main point of contention. The S\_TREQST is the message format used by the client to request for edit permissions to the individual section objects of the document.

The *Section ID* field is a 32-bit unsigned integer. The Section as denoted above is the Section for which edit permissions are requested by the client to the server. We only allow for 1 section request at a time. Intuitively, this makes sense since we are most likely to edit an individual section (object) at a time.

The granularity of the section can be arbitrary. If the granularity of the section in question is known by the application, then a *length* field can be used here. The length field denotes whether we want a section for a specific length from the start or say we want to start a new section at the end of the document.

If the case is the latter, then we can add a *V* bit for the section to denote our request type as variable length section request. The variable length field is important since it requests the application for higher privileges to the section edit.

Another bit is the urgency bit, *U bit.* The urgency bit can be used for scheduling. Suppose our application is well behaved and clients adhere to the rules of the protocol, then a request from a client with higher urgency who is currently editing the documents can be treated with more (or less, depending upon starvation of requests and fairness) importance than fresh requests for the section.

The bits *R1-R29* are treated as reserved here for adding additional such request options for finer detail of privileges required by the client and application level control required by the server.

*In our implementation, we choose not to implement variable length sections so the V bit is 0 and also the urgency bit so the U bit is 0 here. (dealing with static length documents)*

**Request = “S\_TRESPN”**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Representation** | **Byte Location** | **Description** |
| token | 64-bit double(long) | 40..47 | 64-bit double value for token |
| Section | 32 bit integer | 48..51 | 32-bit unsigned integer value for section number |
| Length | 32 bit integer | 52..55 | 32-bit unsigned integer length field for section number in request |
| Additional Privileges | 64-bit unsigned integer | 56..63 | 64 bits additional reserved field for additional privileges |

The *S\_TRESPN* is the response from the server to the client request if the server agrees to grant the client a request for the token. The token here is an encrypted value which if used independently holds no significance. As we shall see, that this token is logged by the server whenever the client wants to push his updates to the server. This might seem counterintuitive since why would be want to acquire a token then in the first place. But the point to note here is that this reduces the point of contention just like caching or registers in CPU’s. If a commit has to take place, it the client has to supply this token back and then wait for the server to respond. At any point of time here, the server can issue a S\_REVOKE to invalidate the *token (timer – based revoke in our design and implementation*. This is significant since only 1 token for a section is out on the network at any given moment of time.

The 64-bit double value for a token signifies a random number bound to a specific section by the server at a particular instant of time when the section was requested.

The Section number indicates the section for which the token was requested. This is just used to validate the section request on the client side so that the client does not get a response for which it has posed no request.

The length field associates the length of the section.

*Since we are using static lengths in our implementation, we are using 0 for this field in our implementation.*

We have kept an *additional privilege* field vacant to be accommodated for some other critical instructional feature to be used by the client-level application as to how to use the token. For instance, the token expire itself in some predetermined period of time. It is just an extensibility feature specific to the application using the protocol.

The point to note about this PDU is that this PDU is not variable length. In fact, all such frequent length PDUs have been worked in such a way so as to maintain the controlling information pushed by the server to a minimal size.

*Status Messages:*

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  | **Response = "S\_DENIED/S\_REVOKE/S\_DONE"** |
|  | **Representation** | **Byte Location** | **Description** |
| Status Code | 32 bit unsigned integer | 40..43 | Status Code |
| Error Reason | 32 bit | 44..47 | reason for error if any 0 otherwise |

S\_DENIED, S\_REVOKE and S\_DONE unlike blank PDU’s are status codes which also supply a reason to be accommodated if any. *S\_DENIED* as the name suggests is a server to client message to inform the client that his request using S\_TREQST cannot be served at this point. A reason can be attached in the PDU.

This makes sense because the client can then stop sending further updates to the server.

*S\_DONE* is when the server has correctly processed the commit by the client. The error reason for this is definitely 0.

*S\_REVOKE* is when the server revokes the token back from the client. A revoke here means to revocation of rights which the server simply implements by invalidating the client token and sending a S\_REVOKE message.

*In our design and implementation, S\_REVOKE means that the token has been revoked due to a timeout.*

*Please Note:- The status and error codes aren’t implemented by us since we have unique interpretation of these fields at this point. A more dense implementation can choose to implement these codes with multiple reasons for a denial for example.*

4

**Request = Request = "S\_COMMIT"**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **Representation** | **Byte Location** | **Description** |
| E(token+privileges) | 128 bits | 40..55 | Echo back the token sent by server/ Server can either invalidate this token or extend its lifetime by sending another S\_TRESPN |
| Previous Section ID | 32 integer value | 56..59 | The ID of the previous section |
| Section ID | 32 integer value | 60..63 | The ID of the section to be committed |
| Length | 32 integer value | 56..59 | Length of commit |
| Data | 8bit character | 60..<END> | Data |

As discussed in the PDU above, the S\_COMMIT is the PDU from the client side to be able to push his/her commits (section edits) back to the server so that the server can process them accordingly.

The client just echoes back his received token+privileges but the *E (token+privileges)* level applies to the client side encryption so that the server can validate that the update is from the client which received and decrypted the token supplied earlier. The length is the same 128-bits for reasons mentioned above.

The other fields here are *length and data*. Since the client wishes to push his updates back to the server, the client here uses the length field to denote the length of his update (edit) and the associated data which has to be committed to the server document from the section ID of the document supplied earlier.

The Section ID field is used to indicate to the server which section the client is attempting to edit, but in the case of a new section the server needs to understand where the client intends to locate this new section in the document. To handle this a Previous Section ID field indicates the section that should be prior to this section (Note, Previous Section ID = 0 indicates the start of the document)

**Miscellaneous PDUs**

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| “ABORT” PDU | |  | |  |  | | |
| Field | | **Representation** | | **Byte Location** | **Description** | |
| none | | none | | none | none | |
|  | |  | |  |  | | |
| “ECHO” PDU | |  | |  |  | | |
| Field | | **Representation** | | **Byte Location** | **Description** | |
| none | | none | | none | none | |
|  | |  | |  |  | |
| “BLOCK” PDU | |  | |  |  | |
| Field | | **Representation** | | **Byte Location** | **Description** | |
| username | | 20 character ASCII | | 40..59 | username to block | |
| requested block type | | 2 bits | | 60(bits0&1) | request account-block (11), document-block (10), time-block (01), or server-decided block (00) | |
| malicious user flag | | 1 bit | | 60(bit2) | suspected of malicious behavior | |
| impersonator flag | | 1 bit | | 60(bit3) | suspected of account-impersonation/stealing | |
| removed user flag | | 1 bit | | 60(bit4) | user removed as collaborator on document | |
| Over-editing flag | | 1 bit | | 60(bit5) | user editing too much and preventing other users from editing | |
| bot flag | | 1 bit | | 60(bit6) | user suspected of being/using a bot | |
| Sock-puppet flag | | 1 bit | | 60(bit7) | user suspected of being a sock-puppet for another user | |
| reserved flags | | 3 bytes | | 61..63 | flags reserved for extension | |
|  | |  | |  |  | |
|  |  | |  | | |

Some messages have no extra content. These messages include ABORT, for sudden disconnections caused by some error, LOGOFF, for gentle, user specified disconnection, and ECHO.

The BLOCK message format starts with the 20 8-bit ASCII character username of the individual to be blocked. The next 4 bytes consist of flags detailing the request. The first flag is 2 bits and identifies the type of block being requested. The requested block can be for the entire account (represented by 11), for access to the specific document (10), a time-based block (01) or server-decided block (00). All implementations must allow for the server-decided block. It is up to the given implementation whether to honor the other three versions. The remainder of the first byte of flags are single bit flags for various reasons for requesting a block. They are, in order, suspicion of being a malicious user, suspicion of stealing/impersonating someone else’s account, removal of the user as a collaborator on the document, preventing other users from editing by editing too much or for too long, suspicion of being or using a bot, and suspicion of being a sock-puppet, or dummy account. The remaining 3 bytes of flags are reserved for extension, as need for more block request details may become apparent overtime.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| “LOGOFF” PDU |  |  |  | |
| Field | **Representation** | **Byte Location** | **Description** |
| none | none | none | none |

LOGOFF is just a control signal sent by the client when the client wishes to close his connection. The server just responds with a LACK (LOGOFF Acknowledgment)

|  |  |  |  |
| --- | --- | --- | --- |
| “LACK” PDU |  |  |  |
| Field | **Representation** | **Byte Location** | **Description** |
| none | none | none | none |

LACK as discussed above is the Server side acknowledgement for client LOGOFF. No information is kept in this packet as well. It is just an informational status message that the server acknowledges this.

**Protocol DFA**

***C:\Users\Anthony\Downloads\DFAfinal .png***

***Construction Details:***

Let me first mention that the DFA above is encoded as per various PDUs we have developed for our RTCE protocol. The DFA follows the standard formalism where messages/events received cause a state transition followed by / (separator) to denote the actions which must be taken in response to the status change.

The solid lines strictly denote the state transition for the client. The server side transitions are denoted by dotted lines. The DFA is constructed to be as comprehensive as possible and can be considered as complete in its scope as far as the PDUs are considered.

The red lines are just an ABORT PDU sent by the client in case the client does an ungraceful abort of the active connection/handshake.

***Transition flow details:***

The client and server are initially in the CLOSED state. The server is IDLE when it is active and then is considered active throughout its lifetime. The client enters from the CLOSED to the CONN\_PENDING state when its first sends a CUAUTH to the server to authenticate its details and in short serves as an initialization of the handshake from the client end. The server recognizes the client and then sends its own authentication details through a CONNECT PDU.

Once the client receives a CONNECT, the client enters the CONNECTED state and responds with a CACK. Here, CACK stands for connection acknowledgment from the client end. The server although does not enter the CONNECTED state unless it does not receive the CACK response from the client. The client can choose to abort the connection or respond to connect to the server at this point, *although our implementation automatically sends a CACK from the client once it receives a connect from the server.* Our design and implementation does not deal with half open connections at this point although a timeout to discard the connection would be the standard way to do it.

Once connected, the client waits for the server to respond with the initial list and data associated with the list of sections. These 2 messages are separate and the S\_LIST precedes the S\_DATA message in the ordering. The client enters the READ\_DOC state only after receiving both these kinds of associated messages. The client stays in the read mode as long as it chooses to stay there before asking for edit permissions from the server.

The client if at any moment in this state chooses to avail permission to make edits to the document, it enters a different state which is WRITE\_DOC, and it sends a request for permissions/updates to the server through a S\_TREQST. The S\_TREQST is needed to request for permission to edit specific sections /section of documents. The server responds with a token S\_TRESPN to the client as permission to edit and then only the client can enter the WRITE\_DOC state.

If the permission is denied, then a S\_DENIED control signal is sent to notify the client that their requests have not been granted. Hence, the client again enters a READ\_DOC state. These states are separate for permission control and effective communication here.

*In our implementation, the client can only request one token at a time and the server can allocate a token to a client if it currently has no token and the section in the document for which the request is made is free. This prevents us from the obvious malicious attack to the design in which the client requests for multiple tokens to block out other people to access the corresponding sections in the document*.

If the client chooses to make edits, he can commit his edits to the document by sending a S\_COMMIT message and wait for a done response from the server. When the server responds with a done message, it means that the client updates have been successfully committed to the document. If the token matches, then a S\_DONE message is sent to the client. To ensure that the client cannot cling onto a token for a long time, we have a S\_REVOKE message. If the timer associated with the client token expires then the token is revoked from the client.

*In our implementation, the client can commit changes to the document only if the token matches with the current binding of the token to a section of the document. If the binding of the token to a section does not match on the server side, it means that the section is in use after it has been revoked from the client and the commit could be from a malicious user.*

This brings us to a new feature with respect to our protocol which is BLOCKED state. The blocked state is a state which the client can enter if he/she has gone rogue and/or an administrative authority chooses to block one or all clients. This can be permanent or temporary (temporary if say the owner of the document wants to make an important change).

In our protocol design we have chosen to do a temporary block in which case we again chose a timeout-based block. In case of a timeout based block, the client is again allowed to continue his session after the timeout expires. This features automatically accommodates to block a client at any state in read, write or connected after he is connected. If a client is blocked, the client can still send requests but the server will not process them. The client is only notified at 2 stages, one when it is blocked and it tries to send a request, it will be notified that it is blocked. Secondly, when a client is unblocked (due to timeout), the server will send an echo back to the client.

*Specific to our implementation, the owner of the document is by default the one who is the first person to log in and has the authority to block a client. The client is simply blocked by the owner until the preconfigured timeout. The logic for doing this is it gives the owner a window to make important changes as well as to do away with blocking out someone indefinitely. If the client is unblocked, then all the pending requests of the client are processed if they can still be met. If not, the client is notified of the same.*

The final stage in the communication process is the Logoff. The Client sends a LOGOFF to the server and waits for an LACK (acknowledgment) from the server which is the server side logoff acknowledgment. When the server responds, the client simply disconnects here. The server can also switch to the IDLE state again if it has no active connections pending in its queue. This makes sense here since the RTCE demands many users to access documents at once and then completely not. It means the flow is in bursts of when multiple clients try to access a single document and truly collaborate in the environment.

A distinction is made between ABORT and LOGOFF. The abort message is when the client does not need to wait a response, acknowledgment from the server. This makes sense here. Say, in our protocol say for some reason the server is temporarily down or if it has many multiple requests to server. The client in an emergency can just do an ABORT and the connection /handshake close is notified to the server.

*In our implementation, if the client is in the write mode and he has a token then before a log off or an abort the server automatically revokes the token from the client.*

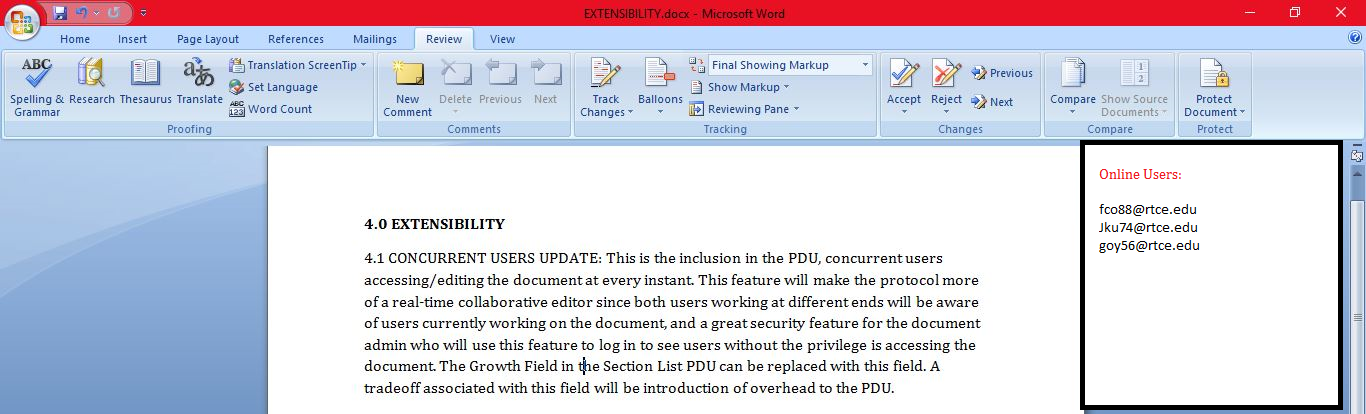
*Therefore, the crux of our protocol is with respect to token management which we ensure is not blocked out by a client by implementing timeouts, revokes and one token at a time mechanism.*

**Extensibility**

The following fields were added to these PDUs for extensibility:

* Header PDU: The reserved fields (Reserved1, Reserve2, Reserve3)
* CUAUTH PDU: other options field
* CONNECT PDU: other options field
* Section-List PDU: Growth field (Growth1 - GrowthN)
* S\_TREQST PDU: R1 – R29
* Block PDU: Reserved flags.

A new message which can be added to the later versions is the concurrent users accessing/editing the document at every instant. This feature will make the protocol more of a real-time collaborative editor since both users working at different ends will be aware of users currently working on the document, and a great security feature for the document admin who will use this feature to log in to see users without the privilege is accessing the document. The support of this option can be indicated in the other options field of CUAUTH & CONNECT PDU or supported simply by being on a particular version established in the authentication process. A tradeoff associated with this message will be introduction of additional messaging in the network. Below is a notional picture.



*The sidebar notifies the user of the presence of other users currently accessing/editing the document.*

**MIGRATION**

A major concern with the introduction of a new version of network protocol is backward compatibility. The version field in CUAUTH and CONNECT is used to advertise the protocol’s version solves this problem. When conversation is to occur between two RTCE protocols of different version, the version field ensures that the minimal common theme supported by both protocol is used.

For example a RTCE version 1 client trying to communicate with RTCE version 1.1 server, the client uses CUAUTH to contact the server, the RTCE server looks at the version number and uses the minimal common theme supported by both protocols.

**Security**

We consider security to be an important but secondary element of our protocol. To this end, every packet header includes a session identifier, and once a session terminates it cannot be resumed. Sessions are established during a security handshake at the beginning of the session. The first message from the client application to the server includes both authentication information and a list of session options. In particular, encryption options have their own category. The server’s response includes a server identifier and the option selections. Use of those options starts with the client’s acknowledgement of the completed handshake.

The most basic security measure is the authentication. We require users to authenticate to use the service. By default, we assume username and password authentication. However, we have built in generic options in the handshake which could be extended to include alternate authentication techniques. The server also sends an authentication key. If the sent key is not trusted, the user is recommended to change their authentication information when they successfully connect to the correct server.

Another basic security measure is encryption. By default, we do not require encryption, but we provide support for implementing and selecting various encryption options. Similarly, we could use our generic options to extend the protocol with other security mechanisms. Furthermore, we included reserved fields in the header for extension, and one extension we propose is random padding, to protect against traffic analysis.

Furthermore, we have an extensible set of messages and therefore an extensible DFA. While by default we provide basic security settings and simple extension to include more, we also recommend extending the protocol to include a longer and more secure handshake, and indicate this extension with either a generic option or in versioning.

Additionally, we provide access control in the form of tokens.  We divide the document into portions, the size of which is implementation dependent.  For each portion, only one person may edit at a time to avoid editing collisions.  To do this, edits are done by requesting a token, and once the token is received there is a window in which to replace the identified section with new text, or to create the identified section if it does not exist.  No other edits may be made without requesting other tokens.  Only one token can be held per client and only 1 client can hold a token for a given section.

**Changes from the initial protocol design paper:**

* Did not need to use a separate UDP interface for the transfer of data sections as suggested in the original paper. This provided to add little/no value for the complexity.
* Originally there was no notion in the protocol of how sections are ordered to form a document. The S\_LIST message took on a new meaning to convey the order of sections to a client.
* Likewise, the S\_COMMIT message had no way to indicate where a new client added section should exist in the document, thus a Previous Section ID field was added to specify where the section should be located relative to another section ID.
* The DFA and its description is changed
* PDUs S\_TREQST , S\_TRESPN fields have changed
* The meaning and usage of status control messages have changed as per the DFA.
* Tokens now are consumed when performing an S\_COMMIT. The original paper allowed the user to keep making multiple edits (S\_COMMIT messages) with reusing a single token.
* While not a “change”, our server implementation did show a performance implication where we create a new thread server-side for each client that logs in to manage their session. This was sufficient for our testing, but would not be sufficient in a larger arena with thousands of clients connecting into a server (creating thousands of threads). We anticipate several ways that one could handle this at the server application level. One possibility is server discovery may be extended such that “full” servers will not reply back to be discovered, or even load-balancing such that only the least loaded server in a multi-server network may respond back to client discovery attempts.