Group 2: The Remote Automobile Utility Control Protocol

CS544 Spring 2017, Drexel University

Lewis Cannalongo
lrc74@drexel.edu

Max Mattes
mm3888@drexel.edu

Ismail Kuru
ik335@drexel.edu

June 9, 2017

Abstract

We propose an application layer protocol for remote automobile utility control, RAUC. Automobiles are getting more and more complex and there exists a great shift in controlling cars with softwares. Softwares are designed for controlling utilities in an automobile such as radio-controller, ac-controller, side-mirror-controller and car-internals-report etc. Smart devices are getting more and more popular and they are used as a tool in many domains with help of domain specific softwares where controlling vehicles is one of them. Our protocol enables a remote user, the client to connect to a centralized server, for controlling his one of vehicles utilities with his smart device. In this paper, we explain the protocol, RAUC, from many aspects including:

- Client and Server communication through deterministic finite state machine, Protocol-DFA.
- Common themes : Addressing, Flow Control, PDU-Message-Definition, Error Control
- Extensibility
- Quality of Service
- Security Implications

Contents

1 Service Description

The Remote Automobile Utility Control **(RAUC)** protocol specifies communication between user-controlled devices and the Automobile Control Server **(ACS)** through variable-sized messages sent across a TCP connection with all communications encrypted through TLS. These messages include commands for the ACS to forward to the automobile, and queries against automobile component state. RAUC connections require clients to authenticate themselves with the ACS before any commands or queries may be issued. This protocol assumes the user has already registered their car with the ACS, and that they have a valid user account authorized to access the vehicle in question. It is also assumes that the vehicle is either always connected to the internet, or that commands and queries will be queued until the vehicle is reachable, but the exact details are handled by the ACS and beyond the scope of the protocol itself.

The protocol itself provides a secure and extensible way to issue commands to a fleet of vehicles. The PDUs are flexible enough to allow for query and control of a variety of components and remain extensible into the potential future of car development beyond the different types of components which exist today.

2 Common Themes Detailed

In this section, we discuss the details of the RAUC protocol.

2.1 PDU (Message) Definitions

RAUC uses a single generic, variable sized PDU comprised of two parts: A 2 Byte HEADER CHUNK and a list of up to 255 CONTENT CHUNKs, which each range in size from 1 to 256 Bytes. The PDU is delineated by the end of the final CONTENT CHUNK in the message. Determining the position of the end of the last CONTENT CHUNK is defined below. This same PDU is used for both client-to-server and server-to-client messages. Exact message type codes are listed below in §2.1.1.

MESSAGE: [HEADER CHUNK : [CONTENT CHUNK] [...]]

The HEADER CHUNK is composed of 1 Byte specifying the type, and 1 Byte specifying the number of following content chunks. RAUC supports up to 255 unique message types (with 1 reserved for future extension) and the number of content chunks in a given message can range from 0 to 255.

HEADER CHUNK: [MESSAGE TYPE : CHUNK COUNT]

CONTENT CHUNKs are variable size. The first Byte specifies the number of Bytes to follow it with data for that chunk. Each following chunk may contain data in UTF-8 encoding. There should not be any padding bytes since the size is specified exactly.

CONTENT CHUNK: [SIZE : DATA]

2.1.1 Message Types

In table 1, we list the message types and their usage. Each message type has a number of required content chunks that form the parameters of the message. Below, we go into the details of each message and provide the structure and an example with each type.

Message Type	Message Type Code	Required Content Chunks	Usage
User Authenticatoin	3	2	Client Handshake Message
Authenticatoin Acknowledgement	4	[0 — 1]	Server Handshake Message
Utility Control Request	5	4	Client Command to Server
Request Received	15	[0 — 1]	Server Response to UCR
Utility State Query	6	[0 — 1 — 2 — 3 — 4]	Client Command to Server
Query Result	16	1	Server Response to Query
Permanent Error	255	0	Error Message
Temporary Error	254	0	Error Message
Termination	9	0	Close the connection

Table 1: Message Types

3 - User Authentication : The User Authentication command is a 3-chunk message made up of the header and two content chunks: USERNAME and PASSWORD.

[3:2] : [USERNAME] [PASSWORD]
[3:2] : [5:uname] [17:aVeryLongPassword]

4 - Authentication Acknowledgement : This message lets the client know that they are authenticated and the server is ready to begin receiving Utility Control Request or Utility State Query messages. ACS servers may use Authentication Acknowledgement commands to transmit additional information such as protocol version, supported extensions with an extended content chunk count.

```
[4:0] : []
[4:1] : [SERVER VERSION]
[4:1] : [1:0]
```

5 - Utility Control Request : A Utility Control Request (UCR) represents a command to be executed by the vehicle. A UCR contains four content chunks: AUTOID (unique identifier of the automobile), COMP (the specific component on the automobile to access), ATTR (the attribute of the component we wish to modify), and VAL (the value we wish to set the attribute to).

```
[5:4] : [AUTOID] [COMP] [ATTR] [VAL]
[5:4] : [5:34769] [5:radio] [6:volume] [1:5]
```

15 - Request Received : A Command Received (CR) (header [15:0-1]) message is sent to the client when the server successfully receives a Utility Control Request command. The message does not contain any information about whether or not such commands were forwarded; only that the command was valid and accepted by the server. Command received messages may contain 0 or 1 content chunks. ACS servers may send 1-byte content chunks to verify whether or not a command was successfully forwarded to an automobile. A content chunk 0s indicates failure, anything else indicates success.

```
[15:0] : []
[15:1] : [SUCCESS]
[15:1] : [1:1]
```

6 - Utility State Query : A Utility State Query (USQ) message is for requesting information about a vehicle. This command has 0 to 4 CONTENT CHUNKs depending on the nature of the query.

If no arguments are provided, the server returns a list that uniquely identifies each automobile which has been registered to this account that is currently connected to the ACS server. This list essentially provides a list of automobiles that the client can interact with at the present time.

The AUTO argument (content chunk 1) is the unique identifier for a specific automobile which this account has registered to itself (i.e. from the previously mentioned list). If this argument is provided, the server returns a list of all component identifiers which the client may interact with on the specific automobile.

The COMP argument may only be provided if the AUTO argument has been provided. COMP is the identifier which identifies a specific component on AUTO. If this argument is provided, the server returns a list of all attributes available for modification on the indicated component on the indicated auto.

The ATTR argument may only be provided if the AUTO and COMP arguments have been provided. ATTR is the attribute of the specific component on a specific automobile. If this argument is provided, the server returns a list of all values which are valid for the given attribute on the specific component.

The VAL argument may only be provided if the AUTO, COMP, and ATTR arguments have been provided. It is a dummy content chunk that instructs the server to return is the currently-set value of the ATTR on the specific COMP for the specific AUTO provided.

```
[6:3] : [AUTOID] [COMP] [ATTR] [VAL]
[6:3] : [5:34769] [5:radio] [6:volume]
```

16 - Query Result : A Query Result (QR) message is sent by the server in response to a Utility State Query. Query Result messages contain a header chunk and only one content chunk: the result list of a successful query. The content chunk byte-length will naturally vary depending on the result set returned to the client. The following represents a Query Result message for a one-argument Utility State Query with a 64 byte content chunk.

[16:1] : [REGISTERED VEHICLES]

9 - Termination : Termination (header [9:0]) messages may be sent by both client and server. Once command for termination is sent by any of two, all resources being used for the connection is properly finalized and connection is terminated.

[9:0] : []

2.2 Error Control

RAUC supports two types of error messages: Permanent and Temporary.

Permanent Error : (header [255:0]) Permanent errors are sent by the server to the client and instructs the client that the same message sent at the current protocol state will never be accepted. Permanent errors may be sent under these conditions:

- Malformed message received from client.
- Information provided with User Authentication command not verified.
- Message received for an invalid command (eg. User Authentication message sent at any point after authentication handshake)
- Incorrectly addressed message (eg. attempting to issue a command to an unregistered vehicle)

[255:0] : []

Temporary Error : (header [254:0]) Temporary error messages are sent by the server when a command is not received by the server because of a temporal machine state problem. Temporary errors indicate that the client sent a valid command which can be tried again at a later time without modification. The most common reason that a server will send a temporary error message is due to a lack of resources to fulfill a command sent by the client. They may also be sent by the client if the server sends another message while the client expects the server to be listening or vice versa.

[254:0] : []

2.3 Addressing

2.3.1 General Addressing Concerns

RAUC protocol uses TCP/IP for transportation. It uses stream-oriented channels to send messages with no fixed sizes. RAUC messages are passed only between a RAUC client and ACS server, however the ultimate destination of most RAUC messages sent by the client is an automobile simultaneously connected to the ACS server. For messages of this nature, the first content chunk must contain the identifier of the connected automobile for which the command is ultimately destined. It is the duty of the ACS server to verify that the identifier provided by the RAUC client is a valid one for the current connection (user account). The messages defined by this document which require addressing to an automobile are message types 5 (Utility Control request) and message type 6 (Utility State Query). The secure transmission of vehicle addresses is covered in the Security section of this document. Messages defined beyond the scope of this document which are intended for transmission to an automobile must require vehicle identification and only vehicle identification in the first content chunk of the message. All other messages are implicitly addressed to the ACS server or client only.

2.4 Flow Control

Flow control in RAUC is defined by a one-to-one client-message-to-server-message scheme, with the client initiating the connection and pushing messages to the server. The server acts as a passive listener and only sends messages to the client in response to a client request. Both the client and server must not send more than one message without receiving a response from the other host to preserve this granularity of communication. Therefore, the client is the primary driver of the conversation.

2.5 Quality of Service

2.5.1 Sending and Receiving Strategy

ACS servers should handle multiple incoming connections for RAUC on port 7070. Once a connection is established, RAUC clients and ACS servers may negotiate separate sending and receiving ports via a service extension.

ACS serves must send, at minimum, either an error message or some type of acknowledgement for every message that a client sends. RAUC extensions which rely on multiple messages must have an intervening server message between each client message to maintain this basic protocol rule.

2.5.2 Timeouts

Clients must not send more than one message to a server before receiving a response. However, in the event that a server takes too long to respond to a message (a policy decision), a client has two options.

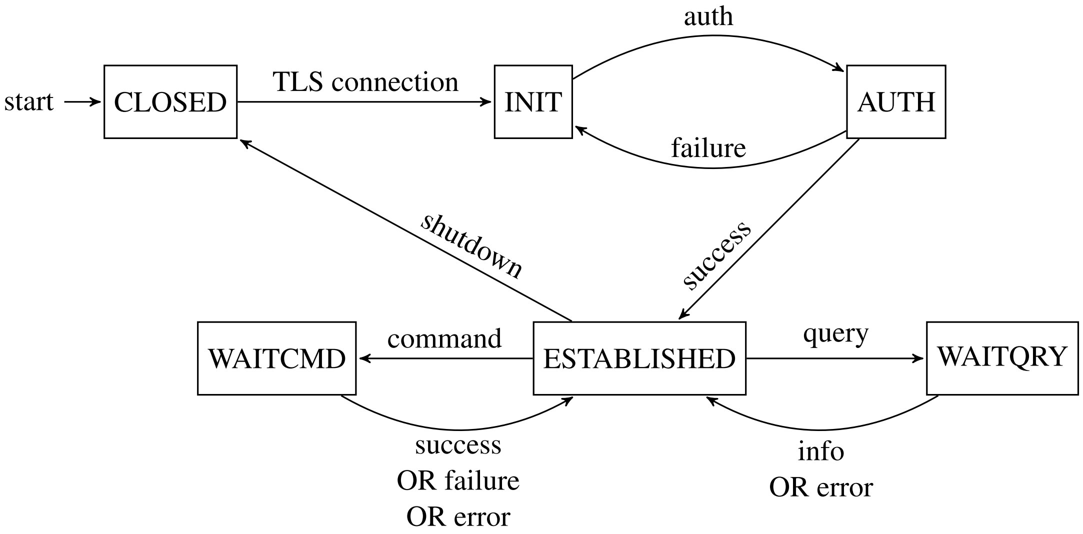
- The client may re-send the message **once**. If this message is accepted, the outcome in this case is the same whether or not it was the original message that failed to deliver or the response: The server will execute the command again.
- The client may end the connection and attempt to reconnect and re-authenticate.

If the client chooses to re-send a message and again does not receive a response from the server, the client must close the connection and initiate a new session with the server.

2.5.3 Performace Enhancement

ACS serves multiple to multiple different clients at a time. ACS provides this enhancement for not serializing other clients' requests when handling one's. We provide this enhancement via designing and implementing ACS as a multithreaded server where a server thread is created for each client connection request so that each client is served by server concurrently.

3 DFA



In addition to the information pictured in the DFA, the shutdown command can be sent from any state. Sending or receiving a shutdown results in termination of the TLS session and moving to the CLOSED state. The implementation can choose whether to send an error in any state where an invalid or malformed message is received, and the recipient can decide whether to send a new message or send a shutdown to terminate the connection. The server should also maintain a `retries` count in the INIT state to count the number of failures sent, and after a configureable `max_retries`, it should sent a shutdown.

4 Extensibility

RAUC employs a service extension model which allow a RAUC client and ACS server to make use of service extensions above and beyond the minimum implementation requirements provided in this document. We do not detail the service extension model in this document but instead provide a high level overview of the model's intended usage structure.

The ACS server, acting as an interface between RAUC clients and connected autos, has more knowledge of what is possible (which autos support specific types of remote control, for example) in a given conversation between a device and a connected auto. As such, ACS servers have heightened responsibility in the end-to-end conversation that RAUC is but one part of. Therefore, the ACS server acts as the moderator for the conversation and is responsible for defining the acceptable version of the conversation and any extensions which may be used throughout.

4.1 Version and Extension Negotiation

Version concordance occurs after a RAUC client authenticates itself with the ACS server. RAUC clients are expressly forbid from issuing commands other than User Authentication during the initial state of the conversation. The Authentication Acknowledgement step therefore provides an optimal time for the server to declare its running version. All ACS servers should return an Authentication Acknowledgement which contains protocol version information, however extended ACS servers must return an Authentication Acknowledgement that contains a list of running extensions. RAUC clients are by default required to accept extended Authentication Acknowledgement messages for precisely this reason and must restrict the messages they send to match those accepted by the server.

4.2 Extension Model

Since all RAUC messages use the same two part HEADER : CONTENT structure, adding new functionality to the protocol is as simple as defining a new message in the client and server applications. The simplicity of the messages also allows for various encodings to be used for content chunks of each message. By default, RAUC clients and ACS servers should use UTF-8 encoding for content chunks, though this can also be negotiated during the handshake.

Currently, the only limit to the model is the fact that the HEADER CHUNK part of each message is limited to two bytes. This limits the number of acceptable message types and size of a given message (in chunks) to 256 of each, though in reality the number of available message codes is fewer due to the minimum required messages of the protocol. However, the size of the message header can change based on protocol version, allowing for future versions to support more than 256 message types that are limited to 65kb in size. In truth, RAUC's strength lies in the simplicity of its message, which is easily scalable by increasing the byte-size of either of any of its subparts.

4.3 Extensibility of Protocol DFA

The generic nature of RAUC PDUs along with the protocol's one-to-one client-message-to-server-message flow control also allows for new protocol states to be easily added. In the same way that the ACS server refuses all messages from the client besides User Authentication during the initial state, the server may limit which messages are valid at any given time based on the protocol state.

For example, consider the popular security option of two-factor authentication. Policy decisions may limit a user's ability to query active components unless two-factor authentication has been used on this connection. This extension essentially places another state after the ”auth provided” state which does not transition to the ”query sequence” state when a Utility State Query message occurs.

5 Security

5.1 Authentication

Authentication is handled by the initial handshake between the client and server. Authentication is done through a username and password. The authentication step occurs after the TLS connection, so the user's credentials are encrypted on the wire and cannot be taken by an outside observer (unless there's some issue with the underlying TLS session). The server must maintain a limited number of retries to prevent brute force attacks. The server should maintain retry information externally to the session and utilize an exponential delay to prevent an attacker from re-intializing the TLS connection and continuing the brute force attack.

The exact authentication mechanism is beyond the scope of this protocol, but we recommend following typical security hygiene such as storing hashed and salted passwords in a secure database, hashing the password immediately upon receiving it from the network, and forbidding transfer of passwords over the network in any form.

5.2 Authorization

If the server receives any requests before the TLS connection, or any requests aside from the Authentication message prior to reaching the `ESTABLISHED` state, it must either discard them or respond with an error or shutdown message. If an error is sent from any state, it must not include any information the user is not authorized to access.

Authorization of users to specific vehicles should be handled by the server application. If a user tries to access a vehicle they are not authorized to access, the server must respond with a nondescript error which does not indicate whether the chosen vehicle exists. Server response to unauthorized access should be identical to attempts to access a non-existent vehicle, including response time, to prevent attempts to acquire information the user is not authorized to know. The server should terminate the connection after repeated attempts at unauthorized access to a vehicle, and follow an exponential delay method similar to failed authentication attempts. Authorization may be split into query-only and full control, in which case the server may choose to respond with a different error for users able to query a vehicle but not control it.

5.3 Confidentiality

User and data confidentiality are established through TLS encryption. If authentication and authorization procedures are implemented correctly, user data should not be transmissible to unauthorized parties, and an attacker should not be able to impersonate a user, and therefore their data is confidential.

Creation and modification of user accounts and permissions are beyond the scope of this protocol. This protocol cannot defend against users who have wrongfully acquired account credentials or permissions they are not supposed to have.

5.4 Availability

Protection against denial of service attacks is beyond the scope of this protocol. Server applications should be careful to implement extra guards against these attacks, especially because TLS overhead can worsen the problem. Servers may choose to refuse TLS connections with users who are on exponential delay due to repeated failed authentication or attempts at unauthorized access to lessen the burden.

The server may also choose to shutdown connections after a certain time period or long periods of silence to reduce load and improve availability. These must be done through the typical procedure of sending a shutdown packet to the client.

The server must cease processing as soon as an error is encountered in a message, to reduce the potential for messages that lie about their size or contain other errors that would utilize CPU time and reduce availability for other clients.

5.5 Integrity

Data integrity is guaranteed by usage of a TLS connection. The sequence numbers and encryption scheme of TLS ensure that repeated and damaged messages will be rejected before reaching the application layer. No messages may be sent outside of the TLS connection to ensure integrity and confidentiality.

5.6 Non-Repudiation

Non-repudiation is also established by the TLS connection. Tight control over authorization and authentication in the server application and refusal of commands prior to authentication ensure the identity of the client before any actions are taken. The usage of sequence numbers in the TLS encryption scheme provides defense against external message replay attacks that could otherwise result in malicious control of the vehicle (e.g. sending repeated commands to raise the vehicle's speed). The client should wait for the server to respond to commands and not re-send them to prevent message replay issues at the application layer.

The server and client should consider adopting a certificate signing scheme similar to HTTPS to prevent identity forgery, thereby preventing unsuspecting clients from sending their credentials to a malicious host.