Group 2: The Remote Car Utility Control Protocol

CS544 Spring 2017, Drexel University

Lewis Cannalongo	Max Mattes	Ismail Kuru
lrc74@drexel.edu	mm3888@drexel.edu	ik335@drexel.edu

May 10, 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 **1**

2 Message Definition – PDU **2**
2.1 Addressing . . . 2
2.2 Flow Control . . . 2
2.2.1 Handshake . . . 2
2.3 PDU (Message) Definitions . . . 2
2.3.1 Minimum Required Implementation . . . 3
2.4 Error Control . . . 5
2.5 Quality of Service . . . 6

3 DFA **7**

4 Extensibility **8**

5 Security **9**
5.1 Authentication . . . 9
5.2 Authorization . . . 9
5.3 Confidentiality . . . 9
5.4 Availability . . . 9
5.5 Integrity . . . 10
5.6 Non-Repudiation . . . 10

1 Service Description

The Remote Automobile Utility Control **(RAUC)** protocol specifies communication between smart-devices and Automobile Control Server **(ACS)** machines through variable-sized messages sent across a TCP connection with all communications encrypted through TLS. These messages represent commands that instruct the ACS machine to either forward the command to or perform a query against some automobile connected to the ACS server. RAUC connections require clients to authenticate themselves with the ACS server before any commands or queries may be issued.

ACS hosts act as middlemen between smart-devices and automobiles, translating commands from smart devices and sending them to the utility components (radio, HVAC system, etc.) of connected autos, while simultaneously formatting utility component metadata (radio volume, tuner setting, temperature, etc.) into messages which can be relayed back to connected smart-devices. Thus, ACS hosts implementing RAUC **MUST** implement both RAUC and some other protocol for final transmission to the automobile. Essentially, RAUC provides a unified protocol only for communication between smart devices and ACS servers. Final transmission to the auto is handled by the ACS server as automakers may provide proprietary protocols for such networked communication. This document contains no specifications for communication between an ACS machine and an automobile.

2 Message Definition – PDU

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

2.1 Addressing

RAUC protocol uses TCP/IP for transportation. It uses stream-oriented channels to send messages with no fixed sizes. [Note: We can customize port numbers to for specific task done by a device on the car. For example: We may want to assign base_port_number + device_port_index ex: 1010+1 = 1011 port no for air conditioner.].

Based on port configuration we explained, a TCP connection is established to the CCS IP address and port number provided.

2.2 Flow Control

Flow control is dispatched to TCP/IP layer. [Note: As future extension, we may mention that we can enhance the flow control with adding asychrony in communication such as when client sends a message and returns but sender send feedback later on. This reduces the congestion of network. In addition, we may buffer requests and flush them to server to save reduce communication overhead for each command.]

2.2.1 Handshake

All the handshaking is handled with [TODO]

2.3 PDU (Message) Definitions

RAUC uses a single generic, variable sized PDU comprised of two parts: A single HEADER CHUNK and an ordered set of up to 255 CONTENT CHUNKs. 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.

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

The HEADER CHUNK is a special, fixed-size chunk that contains two one-byte elements. The first byte specifies the MESSAGE TYPE being transmitted and the second byte contains the number of content chunks (CHUNK COUNT) contained in the message. This means that RAUC supports up to 256 unique message types 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 generically defined and dynamically sized message elements. A content chunk has two parts: SIZE and DATA. SIZE is a single byte that defines how many bytes make up the data portion of the chunk. DATA is the actual data content of the chunk that is meaningful to the application. RAUC clients **MUST** size the data sent in each content chunk it sends out; relying on default implementations (such as setting all usernames to 32 bytes long and setting all unused high-order bytes to 0) is not advised.

CONTENT CHUNK: [SIZE : DATA]

2.3.1 Minimum Required Implementation

All RAUC implementations **MUST** implement the following messages, as well as the two error messages, to be considered fully-functioning. In table 1, we list the message types and their usage. We basically assign a type code to each message type so that we can identify what sort of message is sent/recevived. We structure our messages in a way that we can support variable length messages. We keep track of number of required content chunks of for each message.

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

Table 1: Message Types

User Authentication : After establishing a connection, ACS servers **MUST NOT** accept any commands other than User Authentication(header [3:2]) or CLOSE (header [9:0]) before first authenticating the client. Once authenticated, clients can issue further commands. The User Authentication command is a 3-chunk message made up of the header and two content chunks: USERNAME and PASSWORD. With "..." representing user data, the following is an example of the User Authentication command with USERNAME that is 32 bytes long and a PASSWORD that is 256 bytes long:

[3:2] : [USERNAME] [PASSWORD]
[3:2] : [32: ...] [256: ...]

Authentication Acknowledgement : Authentication Acknowledgement (AA) (header [4:0-*]) commands are sent by ACS servers ONLY after successful user authentication. This message lets the client know that 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. The message header being set to 4 is enough for clients to know that authentication was successful and that the server is listening for commands. Two examples appear below.

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

Utility Control Request : A Utility Control Request (UCR) (header [5:4]) represents a command that a smart device wishes to ultimately forward to an automobile connected to the ACS server. The ACS server **MUST** acknowledge receipt of a Utility Control Request message but **MAY** choose whether or not to acknowledge if the command was successfully forwarded. A UCR is a 5-chunk message made up of the header and 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). Using some sample user data, the following is an example of the User Authentication command:

[5:4] : [AUTOID] [COMP] [ATTR] [VAL]
[5:4] : [red car] [radio] [volume] [5]

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 with a DATA portion of all 1s indicates success while a DATA portion of all 0s indicates failure.

[15:0] : []
[15:1] : [SUCCESS]
[15:1] : [1 : 1111 1111]

Utility State Query : A Utility State Query (USQ) (header [6:0-4]) message alerts the server that the device would like some information returned. This command has 0 to 4 CONTENT CHUNKs depending on the nature of the query, however a hierarchical rule must be followed when providing arguments. The arguments AUTO, COMP, ATTR, and VAL are detailed below.

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.

The following message would provide a list of all radio presets [ATTR] saved on [RED CAR]'s [RADIO] component:

[6:3] : [AUTOID] [COMP] [ATTR]
[6:3] : [RED CAR] [RADIO] [PRESETS]

Query Result : A Query Result (QR) (header [16:1]) 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]
16:1] : [64 : ...

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] : []

2.4 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 in the current 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)

[255:0] : []

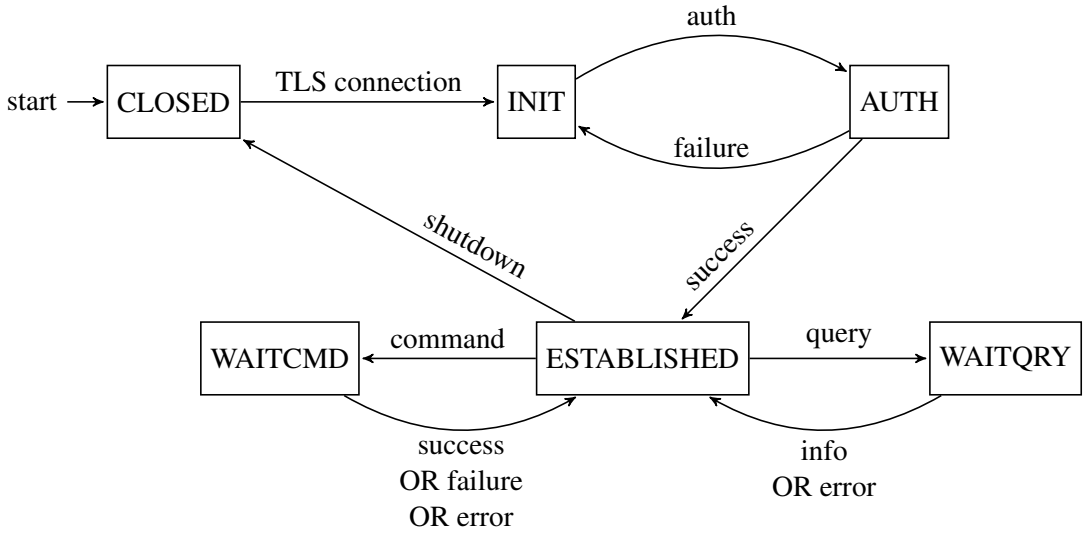
Temporary Error : (header [254:0]) Temporary error messages are sent by the server when a command is not received by the server because of a temporal machine state problem. Temporary errors indicate that the client sent a valid command which can be tried again at a later time without modification. The 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.

[254:0] : []

2.5 Quality of Service

The ... protocol provides several services to the client to ensure that quality is present through its use.

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.

article ulem

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. If a user account has a registered automobile that allows for

remote control of critical components (throttle, brakes - basically anything controlled by a car's ECU), policy decisions may state that NO commands which will flow to an ECU be accepted. Policy decisions may limit a user's ability to query active components unless two-factor authentication has been used on this connection. **needs rewrite: 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.**