Group 12: The Remote SmartHouse Control (RSHC) Protocol

CS544 Spring 2013, Drexel University

Ryan Corcoran
ryan.m.corcoran@gmail.com

Amber Heilman
alh93@drexel.edu

Michael Mersic
mpm76@drexel.edu

Ariel Stolerman
ams573@cs.drexel.edu

May 29, 2013

Abstract

In this paper we propose a protocol for remote smart house control. Smart houses are structures with a centralized control server, that allows controlling various functions and devices in the structure, commonly used for controlling lighting, temperature, security status, entertainment etc. The Remote SmartHouse Control protocol, RSHC, allows a remote user, the client, to log in the house, the server, and perform actions to modify the state of the house, or more precisely its devices. This document details the protocol communication definitions, and analysis of its characteristics, including extensibility, security etc.

Contents

1 Service Description

The Remote SmartHouse Control (RSHC) Protocol serves as a communication mechanism between a client device and server. Built to communicate between a SmartHouse Controller and a remote device, RSHC provides the capability to manipulate devices within the home by interacting with the SmartHouse server, which in turn is responsible for communications with the devices themselves. Though the ability to communicate with household devices is left to the SmartHouse server, RSHC's sole purpose is to translate the actions the client would like to perform to the server so that they may be carried out.

To further explain this relationship, provided is Fig. 1 which shows the direct location of the protocol in relation to the overall SmartHouse schema.

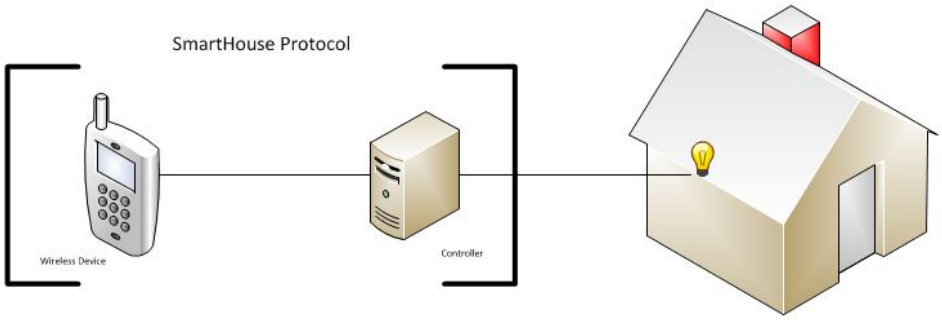

Figure 1: Protocol interaction diagram.

RSHC is independent of a transmission subsystem and requires a reliable ordered data stream channel without fixed size boundaries for transport, for which TCP/IP is the optimum choice for this discussion. An important security feature of RSHC is the use of DES Authentication for the initial authorization of the client device. Although no encryption is provided throughout, the ability to transmit over SSL is a possible option.

The following sections discuss in detail the properties of RSHC, including message definitions, addressing, flow control, security aspects, extensibility and others.

2 Message Definition – PDU

This section discusses the basic definitions of the RSHC protocol. Sec. 2.1 discusses the addressing scheme used by RSHC. Sec. 2.2 briefly discusses flow control. Sec. 2.3 is the main section, and discusses protocol PDU definitions for the different parts of the protocol, including handshake (2.3.2), initialization (2.3.3), and normal protocol communication messages (2.3.4, 2.3.5). Finally we discuss error control in Sec. 2.4 and QoS in Sec. 2.5.

2.1 Addressing

The RSHC protocol is designed to operate over any reliable transport that has no boundaries in data stream (i.e. non-fixed-size messages), therefore TCP/IP is the natural choice. A port enumeration scheme can

be adopted to allow multiple controllers to handle different sets of devices. For instance, one server that provides control over shared rooms in the house (kitchen, dining area, living room etc.) listens to port 7070, another server that is in charge of the master bedroom and its bathroom listens to port 7071 etc.

We choose the base port 7070 as to the best of our knowledge it is not used by any public application as convention. A connection to a RSHC server will then be to the server's IP address, on the respective port discussed above.

2.2 Flow Control

Flow control for the RSHC protocol is handled in the underlying TCP/IP layer, which ensures a reliable message transfer, network traffic moderation and quality of service. However, some factors can be controlled in the RSHC application level: since most of the communication is action-response, where the client sends an action to the server, and the server replies with a confirmation or an action denial (or sends a non-client invoked update), in cases of high traffic an implementation of cumulative messages can be applied. With cumulative messages, several client actions, or several server confirmations / updates, can be aggregated and sent together to reduce network usage. This idea is only brought here as a suggestion for future extension, and will not be supported in the definitions discussed next.

2.3 PDU Definitions

RSHC communication includes 3 stages:

1. *Handshake.* Agree on protocol version and conduct authentication.
2. *Initialization.* Server sends an `init` message to the client with control information.
3. *Normal Protocol Interaction.* Client sends action messages at will, and receives responses from the server. All messages begin with a type (1 byte) followed by message-specific data: client queries or actions and server replies or confirmations.

All messages are constructed of a stream of bytes, either of a fixed size based on the message type or a custom size indicated in the message (which messages are of fixed size and which are of varying size is detailed in the rest of the section). Throughout this document, PDU chunks are formatted as: `[`$\langle title|value \rangle$`:`$\langle \#bytes \rangle$`]`, for instance: `[0x02:1]`. PDUs are preceded by either `S` or `C`, indicating these messages are sent by the server or the client, respectively (or both). All message-type byte-codes, which are always the first byte of any message, are detailed in Tab. 1. Next, each phase is discussed with a detailed description of the RSHC message PDUs.

Message Bytecode	Message Type	Sent by
`0x00`	Poke	Client
`0x01`	Version	Server / Client
`0x02`	Error	Server / Client
`0x03`	Challenge	Server
`0x04`	Response	Client
`0x05`	Init	Server
`0x06`	Action	Client
`0x07`	Confirm	Server
`0x08`	Update	Server
`0x09`	Shutdown	Server / Client

Table 1: List of all message-type byte codes.

2.3.1 Implementation Notes

As will be mentioned in Sec. 6, originally we aimed to have messages passed as a stream of bytes. Since all messages begin with a type, we immediately know whether it is a message of a fixed length (like type 1, which indicates a version message, that always contains *exactly* 12 characters) or a message of a varying length, for which a known position in the message itself should indicate the length of the message (like type 2, error messages where their second byte indicates the length of the following). This approach is designed to well defining message delineation, without using a specific delimiter that indicates the end of the message (as we read bytes, we know how much to expect and keep reading until we cut and look at the buffer thus far as a message).

During the implementation, we found it much simpler to use a Java-provided buffered reader, that simply reads newline-delimited strings from an input stream. Using this method of message reading simplified the implementation by a lot, since it saved us the effort of having to maintain a buffer, and read byte after byte, calculate the amount of bytes left in the current message from what is read thus far, and cut the messages at the right point.

It is trivial to see that using bytes is more efficient than strings, e.g. sending the byte 0x00 (8 bits) is shorter than the string representing the number 0x00, "00", which is 32 bits (and can be even larger, depending on the character encoding). Therefore, in order to support future, more efficient implementations that do not use a simple newline-delimited string reader, but really use streams of bytes, we kept the byte structure of the messages, and converted it to string at the very end before sending it, and back from string to bytes at the very beginning after receiving it on the other end. The following example illustrates how a message is passed, assuming each character is 2-byte long:

```
> sender wants to send a message [0x00]      // 1 byte
> message is converted to the string "00\n"  // 6 bytes (3 chars)
-- message is passed to the other end --
> message is received as a string "00" (reader throws the "\n") // 4 bytes
> message is converted back to the byte stream [0x00]           // 1 byte
```

It immediately follows that having non-fixed-size messages include a byte that indicates the total size of the message is redundant - since messages are now newline-delimited. However, to support future byte-based implementations, and as a secondary length check mechanism, we chose to keep these message-length indicators.

In the rest of the document, we treat messages as streams of bytes, and any potential implementation based on this document should follow this spec. Our implementation, however, takes a shortcut by disguising the byte streams as newline-delimited strings, which is used only for simplicity purposes.

2.3.2 Handshake

This phase is designated for determining version and apply user authentication. To initialize the communication, the client pokes the server with the *poke* message:

```
C > [0x00: 1]
```

The server then sends the highest version it supports. The version message consists of exactly one message-type byte plus 9 ASCII characters (18 bytes) in the format "`RSHC xxxx`" where `xxxx` is the zero-padded version. Example for RSHC v1:

```
S > [0x01: 1]['RSHC 0001': 18]
```

The client responds with the decided version, which should not exceed the version supported by the server. The client version selection message is in the same format at the server's version message:

```
C > [0x01: 1]['RSHC 0001': 18]
```

If the connection failed since the server does not support the requested version, it sends an error message which includes the reason and closes the connection:

```
S > [0x02: 1][#err-msg-chars: 1][err-msg: #err-msg-chars (twice as many bytes)]
```

Otherwise, the server sends the client an authentication challenge message, which includes a 16-byte challenge:

```
S > [0x03: 1][random-challenge: 16]
```

The client encrypts the challenge using DES with a preset 8-character user-defined password, which is encoded into a 16-byte response. He then sends it back to the server, preceded by the his username followed by a semicolon (the semicolon indicates that from this point on the message has 16 bytes left to read – the response itself; the username is not allowed to contain semicolons):

```
C > [0x04: 1][username: ?][';': 2][response: 16]
```

If the response is incorrect, the server notifies with an error message and closes the connection:

```
S > [0x02: 1][#err-msg-chars: 1][err-msg : #err-msg-chars]
```

Otherwise, the server responds with an `init` message, which encodes the available devices and controls in the house to be driven by the client. The format of the `init` message is detailed next.

2.3.3 Initialization

The initialization phase consists of a single server message, in a continuation of the handshake process. With a single server message, the client is notified about all the device types, numbers and states, which altogether comprise the "state of the house". After the client receives the server `init` message, it should have all the information about what devices can be controlled.

One of the challenges for RSHC is how to efficiently encode device information. On one hand, most houses can be assumed to include basic devices that should be available for remote control, like lights, air-conditioning or security alarm; these devices can be encoded efficiently, as common information can be encoded into the protocol (i.e. assumed to be known in advance for both sides). On the other hand, customizable controls for uncommon devices are also desirable, such as the ability to control pool water temperature (under the assumption that smarthouses do not often have swimming pools).

In this document we lay out a solution in which several devices are predefined, along with their possible states and operations. These device types are encoded with increasing integers starting at 0. As discussed in Sec. 4, we leave possible future support in custom messages that can be defined by the house (server) for uncommon devices by simply follow the encoding of known device types and continue the numbering (e.g. for a version that supports 5 known device types, they are encoded as 0–4, and the first custom type will be assigned 5).

The first version of RSHC supports 5 known device types. Tab. 2 details these types, along with their numeric code, states, parameters and actions. Actions are followed by the device states in which they are legal (in parenthesis).

Device Code	Type	States	Parameters	Actions
0x00	Light	[0x00:1] – off [0x01:1] – on	dim	[0x00:1] – turn on (0) [0x01:1] – turn off (1) [0x02:1][level:1] – dim (1)
0x01	Shade	[0x00:1] – up [0x01:1] – down	dim	[0x00:1] – put down (0) [0x01:1] – pull up (1) [0x02:1][level:1] – dim (1)
0x02	AirCon	[0x00:1] – off [0x01:1] – on	temp	[0x00:1] – turn on (0) [0x01:1] – turn off (1) [0x02:1][temp:1] – set-temp (1)
0x03	TV	[0x00:1] – off [0x01:1] – on	channel volume	[0x00:1] – turn on (0) [0x01:1] – turn off (1) [0x02:1][channel:1] – set-channel (1) [0x03:1][volume:1] – set-volume (1)
0x04	Alarm	[0x00:1] – off [0x01:1] – on [0x02:1] – armed	(none)	[0x00:1] – turn on (0,2) [0x01:1] – turn off (1,2) [0x02:1] – arm (0,1)

Table 2: List of supported device types.

The server `init` message is then constructed starting with the `init` message type `0x05`, followed by the list of known device types in order (i.e. first lights, then shades etc.) Each device type starts with a byte indicating the number of such devices, followed by their 16-byte names, current states and parameter values (if any). The number of parameters is determined by the device type, for instance light will have one (dim level), TV will have two (channel and volume), alarm will have none etc. The complete `init` message is then structured as follows:

```
[0x05: 1]
[n0=#lights: 1][name0: 16][state0: 1][params0: 1]...
    ...[name n0: 16][state n0: 1][params n0: 1]
[n1=#shades: 1][name0: 16][state0: 1][params0: 1]...
    ...[name n0: 16][state n0: 1][params n0: 1]
[n2=#aircons: 1][name0: 16][state0: 1][params0: 1]...
    ...[name n2: 16][state n2: 1][params n2: 1]
[n3=#TVs: 1][name0: 16][state0: 1][params0: 2]...
    ...[name n3: 16][state n3: 1][params n3: 2]
[n4=#type 4 devices: 1][name0: 16][state0: 1]...
    ...[name n4: 16][state n4: 1]
```

For instance, the following message indicates there are 2 lights – bedroom light turned off and kitchen light turned on (both dim levels set to 5), no shades, no AC, one TV named 'main TV' turned on channel 18 at volume 7, and no security alarm:

```
[0x05][2]['bedroom'][0][5]['kitchen'][1][5][0][0][1]['main tv'][1][18][7][0]
```

The server `init` message concludes the handshake part of the RSHC communication. From this point on, the client sends requests to the server – actions – and the server responds accordingly, or the server can invoke updates of state changes not invoked by the client (i.e. invoked by other clients connected to

the house, or simply people at the house). Next we detail the client and server messages in the normal communication phase of the protocol.

2.3.4 Client to Server Messages

After the connection is initialized, the client can send an action to the house server at any time it pleases, conforming to the state of the house, available devices and their states. It is assumed that the client will handle maintaining information of the state of the house in order to allow only legal messages to be sent. However, should an illegal message be sent by the client, the server is in charge of responding with the respective error message (as seen in the next section).

The client holds a global counter, initialized to 0 and the size of 1 byte, that maintains a cyclic sequence of the actions it sends to the server. This helps maintaining which actions are confirmed and which are erroneous, as the server will reply to each action with the sequence number attached. It is assumed that a byte will suffice, as the client is not able to perform 256 actions prior to any response from the server (which in theory creates a sequence collision). In fact, since the client is allowed to send only one action at a time, and must wait for a server confirmation / denial before the next action is sent, sequence number collision is not possible. However, we adopt the sequence number scheme for good practice, and to support future versions that may allow multiple actions before any confirmation is received back from the server. A client message is constructed as follows:

```
C > [0x06: 1][count: 1][device code: 1][device number: 1][action: 1+]
```

The message starts with a client-action code byte, `0x06`, followed by the sequence number (which will then increase by 1), and encoding of the target device and action. The length of the action is determined by which action is selected from the list of approved actions in Tab. 2. For instance, the action with sequence number 0x6A "set tv #3 volume to 78" is encoded as follows:

```
C > [0x06][0x6A][0x03][0x02][0x03][0x4E]
```

2.3.5 Server to Client Messages

The server can update the client in two cases: 1) the client sent an action request, and the server updates with a confirmation that the action is applied / denied, and 2) update on a non-client-invoked device state change (for instance, someone in the house turned on some light, or another connected client applied some action).

When an action is received from the client, it is checked for legality. An action is legal if and only if:

1. The device code is legal
2. The device number, for the given device code, is legal
3. The requested action is legal at the current state
4. The given action parameters conform to the requested action

It is assumed that after the initialization phase, the client maintains the state of the house, and therefore should have all the information required to determine which actions are legal at any time and which are not. Despite this assumption, the criteria above are checked for any incoming action request. Should an illegal action be received (i.e. an action message that does not conform to the current state of the house), the server sends a denial message with the given action sequence number:

```
S > [0x07: 1][count of confirmed action: 1][0x00: 1]
```

If the incoming action request is legal, the server replies with a confirmation message with the respective action sequence number. This message is identical to the denial message, only ends with "1" instead of "0":

```
S > [0x07: 1][count of confirmed action: 1][0x01: 1]
```

On non-client-invoked actions performed on the house that derive a state change which is monitored by the remote client, the server updates the action in a message formatted similarly to a client action message, only with a different code and no sequence number. This message is as if the server requests an action from the client, to be applied on the virtual state of the house maintained internally by the client. The update message is constructed as follows:

```
S > [0x08: 1][device code: 1][device number: 1][action: 1+]
```

2.3.6 Common Messages

There are two message types that both the client and the server can invoke. First is a connection shutdown, that can be invoked in order to terminate the communication gracefully. Once sent by any of the sides, any pending actions / updates are disposed and the connection is closed. The termination message is:

```
C|S > [0x09: 1]
```

The second type of message, is an error message with a common message content – whenever an illegal message is received from the other end (where illegal means unexpected message at the current state of the protocol). For instance, if a client receives a challenge when it expects a server version, this error is invoked; or when the server receives an action when it awaits a challenge response, this error will also be invoked. The error message is in the same format as described before:

```
C|S > [0x02: 1][#err-msg-chars: 1][err-msg: #err-msg-chars]
```

After the error message is sent, the connection is terminated.

2.4 Error Control

In the proposed communication messages discussed in the previous section, 3 errors are handled in the protocol:

1. Connection initialization error sent by the server after a client protocol version selection message. This error message is sent when the version is unsupported by the server.
2. The client's challenge response in the authentication phase is wrong.
3. A client action request is illegal.

The first two error messages are followed by terminating the connection, i.e. no error handling is performed. The third message does not terminate the connection, but simply informs the client that its action request is erroneous in the current state of the house. For the first version of the RSHC protocol, this error handling is sufficient, however it can later on be extended to smarter handling (for instance allow multiple attempts to authenticate).

2.5 Quality of Service

The RSHCP protocol provides several services to the client to ensure that quality is present through its use. This particular protocol provides a simplicity warranted for future extensibility and version control, allowing backwards compatibility as well. Even further, the use of an authentication mechanism to ensure security of the client connecting to the server during operations is implemented during the initiation stage. Another feature of the protocol sustains the client knowledge of all household device status changes to provide the client with the most current blueprint of the SmartHouse. These services are defined in detail elsewhere, but are the pillar of service quality in which RSHC is determined to provide.

3 DFA

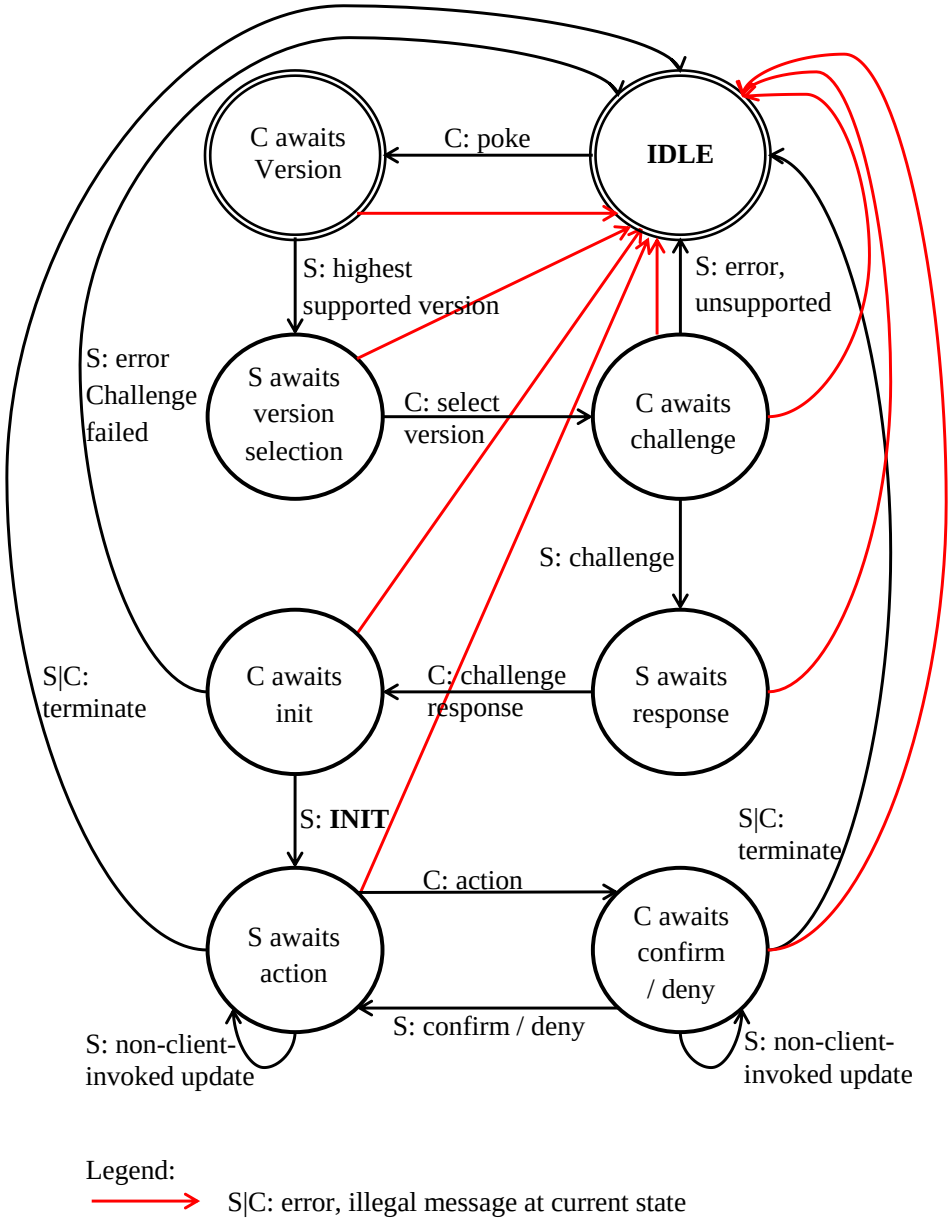

Figure 2: DFA illustration for the RSHC protocol.

An illustration of a DFA for RSHC is shown in Fig. 2. Most of the states consist of handshake and initialization. After the main communication is in progress, there are only two states: the server awaits a client action (bottom left) and the client awaits a server confirmation / denial (bottom right). These two states illustrate the synchronous "ping-pong" communication between the server and the client. In addition, in both states the server may send the client a non-client-invoked action (every time the state of the house changed not due to this client's actions).

We chose to use only two states for the main communication phase of the protocol for simplicity. However, we can pair a DFA state to each subset of legal messages, which means a state for every configuration of the house: the states of all monitored devices. This approach would yield a number of states exponential

in the number of devices, and a correspondence between the states of the protocol to the states of the application. Since the two-states model is sufficient for representation, it is chosen for the illustration of the protocol.

4 Extensibility

Extensibility in RSHC is enabled by the initial version handshake between the client and server.

Additional built-in device types can be added to subsequent versions of the protocol by simply adding additional built-in types to built-in array in the INIT state.

While there are several device types built into the RSHC protocol, it is expected that a subsequent version of the protocol should support device types not built in to the protocol. New device types would be supported by passing a device type description and a list of possible device type commands during the INIT state. With this information the client can manipulate and query a new device just as easily as a built in device. For example, `c5` completely specifies a custom device type: begins with the number of `c5` devices (just like for preset devices), but follows device description, number of states and their description, number of actions and their encodings, and finally actual instances information – device names and current state. Note that we leave action encoding format for future development; however we note that it should include parameter information (number, size and order) and states at which the action is legal (the `action enc. size` preceding the action encoding should manage delineation).

```
[0x03 : 1]
[n0=#type 0 devices: 1][name0: 16][state0: 1]...[name n0: 16][state n0: 1]
...
[n4=#type 1 devices: 1][name0: 16][state0: 1]...[name n4: 16][state n4: 1]
[c5=#type 1 devices: 1]
    // device description
    [device type description: 16]
    // # device states and their description
    [m=#state count: 1]
    [state0 desc.: 16][state1 desc.: 16]...[state m desc.: 16]
    // # device actions and their encoding
    [a=#action count: 1]
    [A0=action0 enc. size: 1][action0 enc.: A0]
    ...
    [Aa=action0 enc. size: 1][action0 enc.: Aa]
    // finally, name of instances of the device and their current state
    [name0: 16][state0: 1]...[name c5: 16][state c5: 1]
```

Since it is assumed the transport layer is reliable and connection oriented, new message types can be added by defining them in a subsequent version of the protocol. No assumptions are made about the DFA that must be carried over to a future version, therefore adding or modifying states is as simple as defining them. Of course, backward compatibility may be a goal of a future version. In that case it is recommended that the only way to enter a new state is with a new message type.

5 Security Implications

5.1 Security

Since RSHC is used to control devices within the users home, security is a critical piece of the protocol. Authentication is controlled through a challenge-response system. In RSHC the server sends the client a 16-byte challenge that the user authenticates by encrypting it using DES with a preset 8-character defined user password. This ensures that only trusted users are granted access.

5.2 Security Issues

By modern standards, DES is considered to be too insecure for many applications, due to the small 56-bit key size. Although we chose to use DES for the RSHC authentication scheme, which is vulnerable to brute-force attacks or potential reply attacks, under assumptions of closed network operation (e.g. control via devices over the house's local secured network) the authentication is sufficient. However, to allow better security also outside a secured network, better authentication schemes should be supported in future versions. Moreover, using secured socket connection (SSL) can ensure security characteristics including confidentiality, integrity etc. As discussed in the previous section, starting the communication with version agreement ensures that future versions can be extended to support new security types seamlessly without harming backwards compatibility.

6 Differences in the Second Version of the Document

This section describes the changes applied in this version of the document, compared with the previous version. The changes are motivated by issues we encountered during the implementation process, and applied to achieve a more complete or sound definition of the protocol, and a simpler implementation.

6.1 PDU Definitions

In the second version of the protocol we applied several important changes to the PDUs:

1. *All* messages begin with a unique byte that indicates the message type, for instance 0 for a poke message, 1 for a version message, 2 for error etc. In the previous version, most messages had a type, except the version message and the client challenge response message. Back then, the choice was motivated by what is used by a real-world protocol (RFB), however we added a type for *all* messages in order to:
   (a) Have a consistent message format
   (b) Be able to easily assert whether any given message at any given state is legal at that state or not
2. In the previous version, after the connection is initialized and the client can send actions to the server and receive confirmations, we allowed the client to send more actions, as it awaits confirmation for older actions he sent. This defined an asynchronous behavior, which we wanted to eliminate. Therefore, now the client is allowed to send only *one* action and must wait for the server to confirm / deny that action. It follows that the server can have at most *one* pending action to confirm / deny for any active client at any given time.
3. As detailed in Sec. 2.3.1, our implementation disguises the byte-stream messages defined in the PDU section as newline-delimited strings. The PDU section still addresses messages as streams of bytes, and only the implementation uses this shortcut to simplify the code (we simply use Java buffered

readers to read complete messages instead of reading byte after byte). The spec in this document is kept as in the original, addressing messages as streams of bytes, since it is the more efficient way, and anyone who wishes to supply a real efficient implementation of our protocol should not take the shortcut we did and follow the spec.

4. In the original `init` message we encoded device names and states, but did not encode possible parameters (like temperature for AirCons or channel and volume for TVs). In the current version, parameters were added to the `init` message, where the number of parameters is determined by the device types (e.g. light will always have one parameter - dim level).

5. In the original protocol, if an action is found to be illegal because it does not conform to the state of the house, an error message is thrown by the server without terminating the connection (i.e. action denial). Now all error messages cause the connection to be terminated, and the server confirmation message is changed to have a parameter: 0 in case the action is denied and 1 in case it is applied.

6.2 DFA

We applied 2 changes in the DFA:

1. Separated the first state (previously: "IDLE, server listens to connections") into 2 states, first is "IDLE" in which the client does nothing and the server listens to connections, and the second is "Client awaits version", where the client is waiting for a version message from the server.

2. Added explicit error messages in all states (red arrows) for any case an illegal message is received by either the server or the client. At such events, the error message is sent to the other side and the connection is terminated (and the states transitions back to "IDLE").

In addition, since we eliminated the asynchronous behavior in the main communication phase (where now the client *must* receive a confirmation / denial from the server before he can invoke another action) and turned it into synchronous, no longer are the client action messages legal at the "Client awaits confirm / deny" state. If the client is at that state, it must wait for the client to send a confirmation / denial to go back to the "Server awaits client action" state. It follows that, at the "Server awaits client action" state, the server cannot send the client any confirmation / denial (since it cannot have a client pending action to process in that state).