# 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 28, 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.

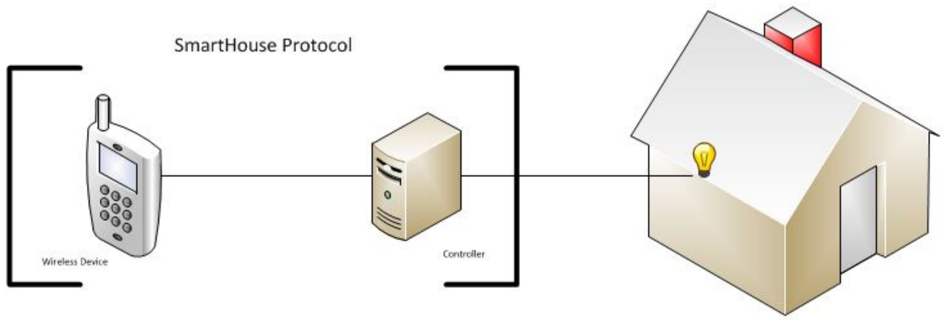


Figure 1: Protocol interaction diagram.

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 (2.3.1), initialization (2.3.2), and normal protocol communication messages (2.3.3, 2.3.4). 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 asynchronous, where the client sends actions to the server, or the server replies with a confirmation or an update, in cases of high traffic an implementation of cumulative messages can be applied. With cumulative messages, several client actions, or several server 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 messages at will, and may receive responses from the server. Normally 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. In the rest of the section, 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). Next, each phase is discussed with a detailed description of the RSHC message PDUs.

#### 2.3.1 Handshake

This is a synchronous phase for determining version and authentication. To initialize the communication, the client pokes the server with the message `[0x00:1]`. The server then sends the highest version it supports, and the client responds with the decided version, which should not exceed the version supported by the server. Each of these messages consists of 10 bytes of ASCII characters in the format "`RSHC xxxx\n`" where `xxxx` is the zero-padded version. For instance:

```
S|C > ['RSHC 0001\n': 10]
```

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 > [0x01: 1][#err-msg-chars: 1][err-msg: #err-msg-chars]
```

Otherwise, the server sends the client an accept message, followed by a 16-byte challenge for authentication:

```
S > [0x02: 1][random-challenge: 16]
```

The client encrypts the challenge using DES with a preset 8-character user-defined password, and sends it in a 16-bytes message back to the server:

```
C > [response: 16]
```

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

```
S > [0x01: 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.

#### 2.3.2 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. 1 details these types, along with their numeric code, states and actions. Actions are followed by the device states in which they are legal (in parenthesis).

| Device Code | Type | States | Actions |
|---|---|---|---|
| `0x00` | Light | `[0x00:1]` – off<br>`[0x01:1]` – on | `[0x00:1]` – turn on (0)<br>`[0x01:1]` – turn off (1)<br>`[0x02:1][level:1]` – dim (1) |
| `0x01` | Shade | `[0x00:1]` – up<br>`[0x01:1]` – down | `[0x00:1]` – put down (0)<br>`[0x01:1]` – pull up (1)<br>`[0x02:1][level:1]` – dim (1) |
| `0x02` | AirCon | `[0x00:1]` – off<br>`[0x01:1]` – on | `[0x00:1]` – turn on (0)<br>`[0x01:1]` – turn off (1)<br>`[0x02:1][temp:1]` – set-temp (1) |
| `0x03` | TV | `[0x00:1]` – off<br>`[0x01:1]` – on | `[0x00:1]` – turn on (0)<br>`[0x01:1]` – turn off (1)<br>`[0x02:1][channel:1]` – set-channel (1)<br>`[0x03:1][volume:1]` – set-volume (1) |
| `0x04` | Alarm | `[0x00:1]` – off<br>`[0x01:1]` – on<br>`[0x02:1]` – armed | `[0x00:1]` – turn on (0,2)<br>`[0x01:1]` – turn off (1,2)<br>`[0x02:1]` – arm (0,1) |

Table 1: List of supported device types.

The server `init` message is then constructed starting with the `init` message type `0x03`, 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 and current states. The complete `init` message is then structured as follows:

```
[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]
```

For instance, the following message indicates there are 2 lights – bedroom light turned off and kitchen light turned on, no shades, no AC, one TV named 'main TV' turned on, and no security alarm:

```
[0x03][2]['bedroom'][0]['kitchen'][1][0][0][1]['main tv'][1][0]
```

The server `init` message concludes the synchronous 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. Next we detail the client and server messages in the normal communication phase of the protocol.

### 2.3.3 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). A client message is constructed as follows:

```
C > [0x04: 1][count: 1][device code: 1][device number: 1][action: 1+]
```

The message starts with a client-action code byte, `0x04`, 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. 1. For instance, the action with sequence number 0x6A "set tv #3 volume to 78" is encoded as follows:

```
C > [0x04][0x6A][0x03][0x02][0x03][0x4E]
```

In addition to action messages, the client and the server can invoke a connection shutdown, 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 > [0x07: 1]
```

### 2.3.4 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, or an error message, and 2) update on a non-client-invoked device state change (for instance, someone in the house turned on some light).

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, the server sends an error message with the respective reason:

```
S > [0x01: 1][#err-msg-chars: 1][err-msg: #err-msg-chars]
```

If the incoming action request is legal, the server replies with a confirmation message to the respective action sequence number:

```
S > [0x05: 1][count of confirmed action: 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 > [0x06: 1][device code: 1][device number: 1][action: 1+]
```

As mentioned in the previous section, the server can also invoke the connection shutdown message.

## 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

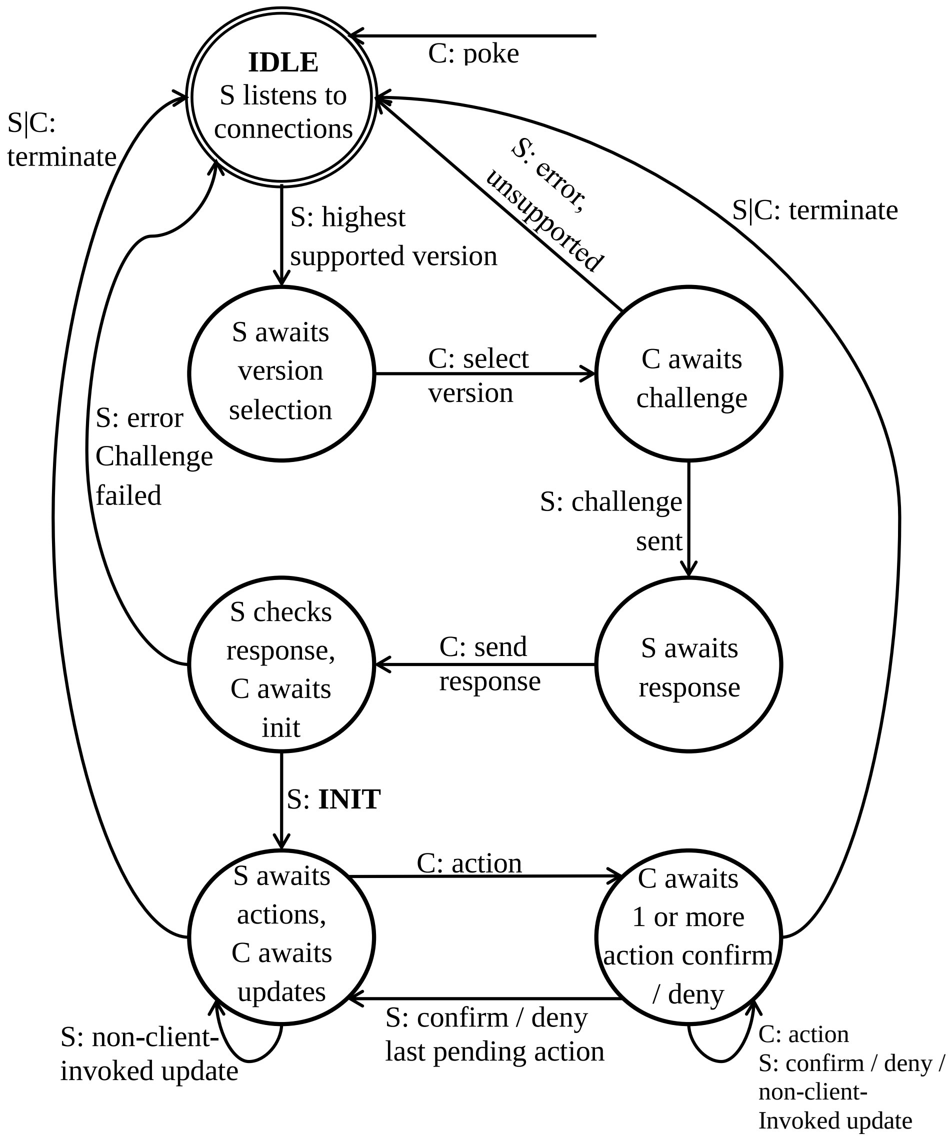


Figure 2: DFA illustration for the RSHC protocol.

An illustration of a DFA for RSHC is shown in Fig. 2. Most of the states consist of handshake and initialization, the synchronous phase of the communication. After the main communication is in progress, the asynchronous phase, there are only two states: the client sends an action and awaits a confirmation (bottom right state), or simply idle (bottom left state), in which the client may receive updates from the server on non-client-invoked actions. Note that when the client awaits a confirmation, it can continue sending actions and receiving confirmations. Only when the last confirmation / denial is sent from the server, the state switches back to idle.

We chose to use only two states for the asynchronous 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 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.