Hot Protocol Version 2 (HotPv2)

Hottentot RPC Framework

Kamran Amini

June 30, 2016

Contents

1 Introduction

This document talks about request and response structures and mechanisms in Hottentot RPC Framework. Purpose of this protocol is to convey Method Invocation request and response. Current protocol is serialization transparent and can convey a method call with arguments produced with different serialization algorithms. In this version, Hottentot's runtimes can only work with internal serialization mechanism.

2 Glossary

SERIALIZATION
A process in which an object turns into a byte array to be transferred using a channel.

STRUCT
A term used for encapsulation of fields related to a specific entity. It is a structure and it will be generated differently for each programming language.

IDL (INTERFACE DEFINITION LANGUAGE)
An IDL is a language transparent to all programming languages which Hottentot supports. IDL can be generated to any target languages supported by Hottentot RPC Framework.

HOT FILE
A file which contains IDL. Hot files usually end with `.hot` extension.

GENERATOR
A tool for generating stub and struct source codes for a target programming language. Currently, generators for C++ and Java languages are available.

RUNTIME
A library for a specific programming language which performs Service and Proxy operations. Currently, runtimes are only available for C++ and Java.

ENDPOINT
Endpoint is a combination of IP address and a port. One service object or many can be bound to an endpoint.

CURRENT ENDPOINT
When talking about a proxy, Current Endpoint is an endpoint which client has used for connecting to service-side.

SERVICE
Service is an object serving method invocation requests.

PROXY
Proxy is an object which produces method invocation and other types of requests and receives the response. A proxy object talks to an endpoint at first and its request will be delegated to a service object in case of method invocation requests.

PROXY-SIDE
A software or library which tries to interact with service-side objects like endpoints, services, etc.

SERVICE-SIDE
A software or library which serves proxy-side requests and generates suitable response.

PUBLIC KEY INFRASTRUCTURE (PKI)
An infrastructure built on top of specific asymmetric cryptography mechanisms and schemes to provide security solutions to known security issues.

CONVERSATION
Transfer of messages between proxy-side and service-side in a Hottentot connection.

3 Protocol V2

Protocol version 2 introduces new features to protocol version 1 in many aspects. Following sections talk about differences and new features in this version.

3.1 Connection Properties

Hottentot connections uses Session layer and Application layer in OSI model. Hottnetot SHOULD use *Secure Socket Library* for *Authentication* and *Encryption* purposes.

3.1.1 Transport

Hottentot SHOULD support TCP for its transport protocol. That's because of long payloads which need sequencing feature in transport layer. Hottentot MAY work with other transport protocols offering sequencing feature and this MAY be up to operating system to handle transport operations.

3.1.2 Type of Communication

Like version 1, communication in protocol version 2 SHOULD start with a request from proxy-side. In this version, service-side SHOULD NOT start the communication. This would be a future feature with its limitations. KeepAlive feature SHOULD BE supported and connection SHOULD remain active by service-side after sending the response if KeepAlive is requested by proxy-side. A Hottentot conversation SHOULD start with **Handshake** messages. Details about the handshake phase will be discussed in next section.

3.1.3 Error Detection

There is no error detection mechanism in this version. We trust underlying protocols like TCP for detecting errors found in the transmitted data. It's obvious that using some security features like SSL provides some tools for detection of communication errors but it is OPTIONAL.

3.1.4 Security

Default behaviour SHOULD use plain transfer of data. This means that security features are optional and they should be enabled by user using configurations. Following sections talk about different aspects of security concerns regarding the Hottentot communication. It is clear that default behaviour does not prevent none of following issues.

3.1.4.1 Eavesdropping

In order to prevent eavesdropping, Hottentot SHOULD use *Encryption.* Symmetric encryption SHOULD be enough but key distribution phase SHOULD use asymmetric cryptography. Any means necessary for implementing the eavesdropping prevention mechanism SHOULD be used in this scenario; examples of such means are certificates, public and private key pairs, etc.

3.1.4.2 Injection

In order to prevent any external party to inject data in a communication channel, Hottentot SHOULD use *Digital Signature* mechanism. A Digital Signature mechanism needs a deployed PKI in order to provide certificates and key pairs for authenticated parties. Hottentot requests and responses MAY carry Digital Signature of request data to assure other parties that data is not changed while transferring.

3.1.4.3 Authentication

In order to provide authentication for parties, Hottentot SHOULD use asymmetric cryptography and PKI compliant certificates.

3.1.5 State Management

Communications SHOULD be stateless. States related to service objects SHOULD be retained from the request payload. Stateful objects are not supported in this version.

3.2 Handshake Phase

As stated before, a Hottentot conversation SHOULD start with *Handshake* messages. Handshakes are performed to let the parties know more about each other and plan for a successful communication. Security objects like Certificates or Encryption Keys are transferred in this phase and the rest of conversation should be set up according to data gathered in handshake phase. Since connections SHOULD start from proxy-side, first message or *Syn Message* SHOULD be sent from proxy-side application. Notice that, this handshake is specific to Hottentot and has nothing to do with other possible handshakes in the system e.g SSL handshakes or TCP handshakes.

3.2.1 Messages

Handshake phase consists of 3 messages. First message or *Syn Message* SHOULD be sent by proxy-side who wants to start the conversation with service-side endpoint. Second message or *Ack Message* SHOULD be sent back by service-side in response to first message. Last message or *Commit Message* SHOULD be sent by proxy-side in order to confirm the conversation or reject it. Once 3 messages are transferred and conversation gets confirmed by proxy-side, request message SHOULD be sent by proxy-side application and it SHOULD wait for the response. Please note that, Hottentot handshake SHOULD be performed after all underlying handshakes. If Hottentot is configured to use SSL, TCP handshake and SSL handshake should be completed before Hottentot handshake takes place. Details about the handshake messages will come in next sections.

3.2.1.1 Syn Message

Handshake Syn Message has following fields:

1. (`START: 0x00, LENGTH: 1 byte`): *Proxy-side Hottentot Protocol Major Version*
2. (`START: 0x01, LENGTH: 1 byte`): *Proxy-side Hottentot Protocol Minor Version*
3. (`START: 0x02, LENGTH: 2 bytes`): *Sequence Number*
4. (`START: 0x04, LENGTH: 1 byte`): *Flag byte*
    - `BIT[0]`: `KeepAlive` flag
    - `BIT[1]` *to* `BIT[7]` *are reserved for future usage and SHOULD be* `0x00`
5. (`START: 0x05, LENGTH: 2 bytes`): *Reserved for future usage and SHOULD be* `0x0000`
6. (`START: 0x07, LENGTH: VARIABLE`): `Array` *of supported* `SerializationAlgorithm`*s*
7. (`START: VARIABLE, LENGTH: VARIABLE`): `Array` *of proxy-side* `Certificate`*s*

If `KeepAlive` flag is set to 1, service-side SHOULD keep connection active after sending back responses. If client is not responsive, the timeout occurs and connection SHOULD be closed by service-side application.

`Sequence Number` SHOULD be generated using a suitable Random Number Generator function. The value SHOULD be used by service-side in order to generate new value for `Sequence Number` in Ack Message. This technique has been used in TCP and it tries to prevent *Packet Injection* attacks.

3.2.1.2 Ack Message

Handshake Ack Message has following fields:

1. (`START: 0x00, LENGTH: 1 byte`): *Service-side Hottentot Protocol Major Version*
2. (`START: 0x01, LENGTH: 1 byte`): *Service-side Hottentot Protocol Minor Version*
3. (`START: 0x02, LENGTH: 2 bytes`): *Sequence Number*
4. (`START: 0x04, LENGTH: 1 byte`): *Flag byte*
    - `BIT[0]`: `Rejected` *flag*
    - `BIT[1]` *to* `BIT[7]` *are reserved for future usage and SHOULD be* `0x00`
5. (`START: 0x05, LENGTH: 1 byte`): *Reason byte*
    - `BIT[0]`: `Unknown or not specified` *flag*
    - `BIT[1]`: `Proxy-side protocol version is too old` *flag*
    - `BIT[2]`: `Proxy-side protocol version is too new` *flag*
    - `BIT[3]`: `Mismatch in set of serialization algorithms` *flag*
    - `BIT[4]`: `Authentication failure` *flag*
    - `BIT[5]`: `Certification not found` *flag*
    - `BIT[6]` *to* `BIT[7]` *are reserved for future usage and SHOULD be* `0x00`
6. (`START: 0x06, LENGTH: 2 bytes`): *Reserved for future usage and SHOULD be* `0x0000`
7. (`START: 0x08, LENGTH: VARIABLE`): `Array` *of supported* `SerializationAlgorithm`*s*
8. (`START: VARIABLE, LENGTH: VARIABLE`): `Array` *of service-side* `Certificate`*s*

If service-side wants to mark the conversation as *rejected*, it SHOULD change the value of `Rejected` flag to 1. If the value is 0, proxy-side MAY assume the conversation as *accepted*. If the conversation is rejected by the service-side, proxy-side SHOULD close the connection instantly.

3.2.1.3 Commit Message

Handshake Commit Message has following fields:

1. (`START: 0x00, LENGTH: 2 bytes`): *Sequence Number*
2. (`START: 0x02, LENGTH: 1 byte`): *Flag byte*
   - `BIT[0]`: `Rejected` *flag*
   - `BIT[1]` *to* `BIT[7]` *are reserved for future usage and SHOULD be* `0x00`
3. (`START: 0x03, LENGTH: 1 byte`): *Reason byte*
   - `BIT[0]`: `Unknown or not specified` *flag*
   - `BIT[1]`: `Service-side protocol version is too old` *flag*
   - `BIT[2]`: `Service-side protocol version is too new` *flag*
   - `BIT[3]`: `Mismatch in set of serialization algorithms` *flag*
   - `BIT[4]`: `Authentication failure` *flag*
   - `BIT[5]`: `Certification not found` *flag*
   - `BIT[6]` *to* `BIT[7]` *are reserved for future usage and SHOULD be* `0x00`
4. (`START: 0x04, LENGTH: 2 bytes`): *Reserved for future usage and SHOULD be* `0x0000`

If proxy-side wants to mark the conversation as *rejected*, it SHOULD change the value of `Rejected` flag to 1. If the value is 0, service-side MAY assume the conversation as *accepted*. If the conversation is rejected by the proxy-side, service-side SHOULD close the connection instantly.

3.3 Request

Each request consists of 1 mandatory and 2 optional fields.

1. (`START: 0x00, LENGTH: 1 byte`): *Request Type*; It is MANDATORY. Using this field determines type of request.
2. (`START: 0x01, LENGTH: VARIABLE`): *Payload*; It is OPTIONAL and carries all necessary data to perform the request operation.
3. (`START: VARIABLE, LENGTH: VARIABLE`): *Request Digital Signature*; It is OPTIONAL and carries a PKI compliant digital signature over Request Type and Payload.

Please notice that, there is no checksum or error detection block involved in request structure. This means that we trust underlying network connection for handling communication errors. Also security concerns like impersonation, eavesdropping or on-wire injection will not be discussed here. There is a dedicated chapter for explaining security issues and detailed provided solutions.

3.3.1 Request Type

Determines the request type and payload structure. Following C enumeration shows the possible values for this field.

```
enum RequestType {
  Unknown = 0x00,
  ServiceListQuery = 0x01,
  MethodInvocation = 0x02,
  MethodListQuery = 0x03,
  ServiceInfoQuery = 0x04,
  MethodInfoQuery = 0x05,
  EndpointInfoQuery = 0x06
};
```

Values can be:

- **Unknown**: It means nothing to Hottentot service-side and these requests SHOULD be ignored by implementation.
- **ServiceListQuery**: Proxy-side queries about the list of available services. Hottentot service runtime SHOULD return list of services exposed on current endpoint.
- **MethodInvocation**: Invokes a method on a specific service object.
- **MethodListQuery**: Proxy-side queries about the list of callable methods on a specific service object.
- **ServiceInfoQuery**: Proxy-side asks about parameters of a specific service.
- **MethodInfoQuery**: Proxy-side asks about parameters of a specific method.
- **EndpointInfoQuery**: Proxy-side asks about parameters of current endpoint.

3.3.2 Payloads

Requests can have payloads. Many request types need data for their operations and payload carries the data. Following sections explain about payload structure for each request type. Please notice that, any needed piece of information or security object for authentication and authorization purposes will not be included in payloads and these objects will be transferred in *Handshake* phase.

3.3.2.1 Payload for *Service List Query* Request Type

A *Service List Query* request has no payload. In other words, this request has only one byte carrying value `0x01` as *Service List Query* request type identifier.

1. (`START: 0x00, LENGTH: 1 byte`): *Request Type*; Value SHOULD be `0x01`.

3.3.2.2 Payload for *Method Invocation* Request Type

A *Method Invocation* request payload consists of *Service Id*, *Method Id*, *Number of Arguments* and *Array of Arguments*. So, the structure of a *Method Invocation* request is something like below.

1. (`START: 0x00, LENGTH: 1 byte`): *Request Type*; Value SHOULD be `0x02`.
2.
   (a) (`START: 0x01, LENGTH: 4 bytes`): *Service Id*; Indicates the service object. See *'Id Generation Algorithms'* section for more details.

   (b) (`START: 0x05, LENGTH: 4 bytes`): *Method Id*; Indicates the method which should be invoked on the service object. See *'Id Generation Algorithms'* section for more details.

   (c) (`START: 0x09, LENGTH: 1 byte`): *Number of Arguments*

   (d) (`START: 0x0A, LENGTH: VARIABLE`): `Array` *of* `Argument`*s*
3. (`START: VARIABLE, LENGTH: VARIABLE`): *Request Digital Signature*; See *'Payload Signature'* chapter for more details.

Arguments SHOULD be transferred using LV structures (See *'LV Structure'* chapter for more details). Arguments are serialized objects so each of them SHOULD indicate the serialization algorithm used for serializing itself.

3.3.2.3 Payload for *Method List Query* Request Type

A *Method List Query* payload has only <u>*Service Id*</u> field. The request structure is like below.

1. (`START: 0x00, LENGTH: 1 byte`): *Request Type*; Value SHOULD be `0x03`.
2.
    (a) (`START: 0x01, LENGTH: 4 bytes`): *Service Id*; Indicates the service object. See *'Id Generation Algorithms'* section for more details.
3. (`START: 0x05, LENGTH: VARIABLE`): *Request Digital Signature*; See *'Payload Signature'* chapter for more details.

This will return list of methods available on a service object identified by *Service Id*.

3.3.2.4 Payload for *Service Info Query* Request Type

A *Service Info Query* payload has only <u>*Service Id*</u> field. The request structure is like below.

1. (`START: 0x00, LENGTH: 1 byte`): *Request Type*; Value SHOULD be `0x04`.
2.
    (a) (`START: 0x01, LENGTH: 4 bytes`): *Service Id*; Indicates the service object. See *'Id Generation Algorithms'* section for more details.
3. (`START: 0x05, LENGTH: VARIABLE`): *Request Digital Signature*; See *'Payload Signature'* chapter for more details.

This will return the parameters and their values for a service object identified by *Service Id*. Parameters and values are serialized according to LV structures (See *'LV Structure'* chapter for more details).

3.3.2.5 Payload for *Method Info Query* Request Type

A *Method Info Query* payload has <u>*Service Id*</u> and <u>*Method Id*</u> fields.

1. (`START: 0x00, LENGTH: 1 byte`): *Request Type*; Value SHOULD be `0x05`.
2.
    (a) (`START: 0x01, LENGTH: 4 bytes`): *Service Id*; Indicates the service object. See *'Id Generation Algorithms'* section for more details.

    (b) (`START: 0x05, LENGTH: 4 bytes`): *Method Id*; Indicates the method on the service object. See *'Id Generation Algorithms'* section for more details.
3. (`START: 0x09, LENGTH: VARIABLE`): *Request Digital Signature*; See *'Payload Signature'* chapter for more details.

This will return the parameters and their values for a method identified by *Method Id* on a specific service object identified by *Service Id*. Parameters and values are serialized according to LV structures (See *'LV Structure'* chapter for more details).

3.3.2.6 Payload for *Endpoint Info Query* Request Type

A *Endpoint Info Query* payload has no field. Parameters of current endpoint (selected by proxy-side) will be returned as response. Parameters are serialized according to LV structures (See *'LV Structure'* chapter for more details).

1. (`START: 0x00, LENGTH: 1 byte`): *Request Type*; Value SHOULD be `0x06`.

3.4 Response

4 Hottentot Serialization

This section explains *Hottentot Serialization* in details. First, we talk about *primitive data types.* Then, we explain the *structs* which they provide a way to *encapsulate* the primitive fields in a single body. And at last, we talk about *nested data types* which they allow us to nest some structs in some other structs.

4.1 Primitive Data Types

4.2 Struct Types

4.3 Nested Data Types

5 Structures

6 Algorithms

6.1 Fast Hash Algorithm

A fast hashing algorithm developed by Zilong Tan. It has 2 versions. One version generates 32-bit has value for a given byte array and the other one generates a 64-bit hash value for the input. Hottentot uses 32-bit fast hash algorithm for generating system independent identifiers (ids) for services and methods. We denote this algorithm with FH_{32} and FH_{64} throughout this document.

$$\begin{aligned} FH_{32} &: \{0,1\}^* \rightarrow \{0,1\}^{32} \\ FH_{64} &: \{0,1\}^* \rightarrow \{0,1\}^{64} \end{aligned} \tag{1}$$

7 Future Features

- Version should be added to request and response structures.