Contract Negotiation HTTP/TLS/JSON

Interface Design Description

Service ID: *"contract-negotiation"*

Abstract

This document describes the HTTP/TLS/JSON variant of the Contract Negotiation service function, which, if implemented by an application system, can be used to negotiate about and enter into legally binding contracts with other systems implementing the same function, potentially located in other local clouds and being owned by other stakeholders.

ARTEMIS Innovation Pilot Project: Arrowhead
THEME [SP1-JTI-ARTEMIS-2012-AIPP4 SP1-JTI-ARTEMIS-2012-AIPP6]
[Production and Energy System Automation Intelligent-Built environment and urban infrastructure for sustainable and friendly cities]

Contents

1 Overview

This document describes the HTTP/TLS/JSON variant of the Contract Negotiation Eclipse Arrowhead service, which is meant to enable distinct parties to digitalize their contractual interactions. Examples of such interactions could be taking on the obligation to deliver a good, negotiating insurance, agreeing to pay by invoice in exchange for access to a particular Eclipse Arrowhead service, among many other.

This document exists as a complement to the *Contract Negotiation – Service Description* (CoNeSD) document. For further details about how this service is meant to be used, please consult that document. The rest of this document describes how to realize the Contract Negotiation (CoNe) service using HTTP [1], TLS [2] and JSON [3], both in terms of its message functions (Section 2) and its information model (Section 3).

2 Service Interface

This section lists the functions that must be exposed by CoNe services in alphabetical order. Each subsection first names the HTTP method and path used to call the function, after which it names an abstract function from the CoNeSD document, as well as any input and output types. All functions are assumed to respond with the HTTP status code `204 No Content` if called successfully, as none of them respond with any data.

2.1 POST /contract-negotiation/acceptances

Function: Accept
Input: Acceptance

Called to accept a previously received Offer, as exemplified in Listing 1.

```
POST /contract-negotiation/acceptances HTTP/1.1

{
  "negotiationId": 55132,
  "offerorFingerprint": {"algorithm": "md5", "sum": "r9y2W1WeiI+6qmMkr/Hz3w=="},
  "acceptorFingerprint": {"algorithm": "md5", "sum": "cmPMoJPtIhUM9tW37sCWvA=="},
  "offerHash": {"algorithm": "md5", "sum": "L43eyod3Sadd42CDe3MpJ+K2BFU="},
  "signature": {
    "timestamp": "2020-06-04T11:42:34.152149Z",
    "scheme": "rsa_pkcs1_sha1",
    "sum": "qITqolHxrZNjdFFvI4t0x+VlJYk="
  }
}
```

Listing 1: An Accept invocation. The ”offer” field has been edited out for the sake of clarity.

2.2 POST /contract-negotiation/offers

Function: Offer
Input: Offer

An offer for the caller and callee to enter into one or more Contracts. An example is given in Listing 2.

```
POST /contract-negotiation/offers HTTP/1.1

{
  "negotiationId": 55132,
  "offerorFingerprint": {"algorithm": "md5", "sum": "r9y2W1WeiI+6qmMkr/Hz3w=="},
  "receiverFingerprint": {"algorithm": "md5", "sum": "cmPMoJPtIhUM9tW37sCWvA=="},
  "validAfter": "2020-06-04T11:39:29.26Z",
  "validUntil": "2020-06-05T11:39:29.26Z",
  "contracts": [{
    "templateHash": {"algorithm": "md5", "sum": "skfh83djseiI+4qmMZ//xz5W=="},
    "arguments": {
      "buyer": "A",
      "seller": "B",
      "quantity": "12",
      "articleNumber": "ABC-123",
      "price": "3750"
    }
  }],
  "signature": {
    "timestamp": "2020-06-04T11:39:29.26Z",
    "scheme": "rsa_pkcs1_sha1",
    "sum": "+O43eyod3SadCDe3MMpJ+K2BFTA="
  }
}
```

Listing 2: An Offer invocation.

2.3 POST /contract-negotiation/rejections

Function: Reject

Called to signal the desire to immediately close an identified negotiation session. An example is given in Listing 3.

```
POST /contract-negotiation/rejections HTTP/1.1

{
  "negotiationId": 55132,
  "offerorFingerprint": {"algorithm": "md5", "sum": "r9y2W1WeiI+6qmMkr/Hz3w=="},
  "rejectorFingerprint": {"algorithm": "md5", "sum": "cmPMoJPtIhUM9tW37sCWvA=="},
  "offerHash": {"algorithm": "md5", "sum": "L43eyod3Sadd42CDe3MpJ+K2BFU="},
  "signature": {
    "timestamp": "2020-06-04T11:42:34.152149Z",
    "scheme": "rsa_pkcs1_sha1",
    "sum": "qITqolHxrZNjdFFvI4t0x+VlJYk="
  }
}
```

Listing 3: An attempt to terminate negotion session 1593831.

3 Information Model

Here, all data objects that can be part of CoNe service calls are listed in alphabetic order. Note that each subsection, which describes one type of object, begins with the *struct* keyword, which is meant to denote a JSON Object that must contain certain fields, or names, with values conforming to explicitly named types. As a complement to the primary types defined in this section, there is also a list of secondary types in Section 3.6, which are used to represent things like hashes, identifiers and texts. Furthermore, as many of these types will have to be used as input to hashing functions, there must exist a canonical way of representing those types. How those canonical forms are derived is specified in Section 3.7.

3.1 struct Acceptance

A signed acceptance of an Offer, being the *candidate* of an identified negotiation session.

Object Field	Value Type	Description
”negotiationId”	RandomID	Negotiation session identifier.
”offerorFingerprint”	Hash	Fingerprint of offeror’s certificate.
”acceptorFingerprint”	Hash	Fingerprint of acceptor’s certificate.
”offerHash”	Hash	Hash of accepted offer.
”signature”	Signature	Acceptor’s signature.

3.2 struct Argument

A concrete Contract parameter value. In other words, this data structure identifies a variable parameter in a template and provides a concrete parameter JSON Value. Conceptually, the parameter specification in the template that matches the *”Name”* field could contain additional type information, such as range restrictions, required list lengths, or regular expressions.

As the *”arguments”* field of the Contract type is a JSON Object, which maps String names to String values, there is no need for a dedicated *Argument* type. This type exists only implicitly and is documented here only for the sake of completeness.

3.3 struct Contract

A parameterized reference to a *template*, representing certain rights and obligations accepted by two parties. In other words, a contract derives its meaning from both the template it refers to and the arguments it contains. Note that contract objects only become binding if they refer correctly to legally valid templates and become part of properly signed Acceptances.

Object Field	Value Type	Description
”templateHash”	Hash	Hash of invoked *template*.
”arguments”	Object<String>	Arguments matching *parameters* defined in *template*.

3.4 struct Offer

A concrete offer of Contracts intended for a specific receiver.

Object Field	Value Type	Description
"negotiationId"	RandomID	Negotiation session identifier.
"offerorFingerprint"	Hash	Fingerprint of offeror's certificate.
"receiverFingerprint"	Hash	Fingerprint of intended receiver's certificate.
"validAfter"	DateTime	Instant when offer becomes acceptable.
"validUntil"	DateTime	Instant when offer expires.
"contracts"	Array<Contract>	Offered contracts.
"signature"	Signature	Offeror's signature.

3.5 struct Rejection

A signed rejection of an Offer, being the *candidate* of an identified negotiation session.

Field	Type	Description
"negotiationId"	RandomID	Negotiation session identifier.
"offerorFingerprint"	Hash	Fingerprint of offeror's certificate.
"rejectorFingerprint"	Hash	Fingerprint of rejector's certificate.
"offerHash"	Hash	Hash of rejected offer.
"signature"	Signature	Rejector's signature.

3.6 Primitives

As all messages are encoded using the JSON format [3], the following primitive constructs, part of that standard, become available. Note that the official standard is defined in terms of parsing rules, while this list only concerns syntactic information. Furthermore, the Object and Array types are given optional generic type parameters, which are used in this document to signify when pair values or elements are expected to conform to certain types.

JSON Type	Description
Value	Any out of Object, Array, String, Number, Boolean or Null.
Object<A>	An unordered collection of [String: Value] pairs, where each Value conforms to type A.
Array<A>	An ordered collection of Value elements, where each element conforms to type A.
String	An arbitrary UTF-8 string.
Number	Any IEEE 754 binary64 floating point number [4], except for *+Inf*, *-Inf* and *NaN*.
Boolean	One out of `true` or `false`.
Null	Must be `null`.

With these primitives now available, we proceed to define all the types specified in the CoNeSD document without a direct equivalent among the JSON types. Concretely, we define the CoNeSD primitives either as *aliases* or *structs*. An *alias* is a renaming of an existing type, but with some further details about how it is intended to be used. Structs are described in the beginning of the parent section. The types are listed by name in alphabetical order.

3.6.1 alias DateTime = String

Pinpoints a moment in type by providing a formatted string that conforms to the RFC 3339 specification [5] (ISO 8601). Naively, the format could expressed as "YYYY-MM-DDTHH:MM:SS.sssZ", where "YYYY" denotes year (4 digits), "MM" denotes month starting from 01, "DD" denotes day starting from 01, "HH" denotes hour in the 24-hour format (00-23), "MM" denotes minute (00-59), "SS" denotes second (00-59) and "sss" denotes second fractions (0-999). "T" is used as separator between the date and the time, while "Z" denotes the UTC time zone. At least three fraction digits should be used, which gives millisecond precision. An example of a valid date/time string is "2019-09-19T15:20:50.521Z". Other forms or variants, including the use of other time zones, is adviced against.

3.6.2 struct Hash

In order for a hash to be verified, it must be known what hashing algorithm it was produced with. As a TLS suite is assumed to be available to implementors of this IDD, hash function names are taken from the TLS IANA standard [6].

Object Field	Value Type	Description
"algorithm"	String	Must be a name defined in [6], such as "sha-256".
"sum"	String	Base64 encoded [7] algorithm output.

3.6.3 alias Name = String

A String that is meant to be short (less than a few tens of characters) and both human and machine-readable.

3.6.4 alias RandomID = Number

An integer Number, originally chosen from a secure source of random numbers. When new RandomIDs are created, they must be ensured not to conflict with any relevant existing random numbers.

3.6.5 struct Signature

In order for a signature to be verified, it must be known what (1) data, (2) public key and (3) hashing algorithm were used to produce it. The first two of those, data and public key, must be resolvable from the context of the signature. In particular, if a signature appears as a field in an Object, the data must be interpreted as being that very object in canonical form, as described in Section 3.7, with the signature field removed. As a TLS suite is assumed to be available to implementors of this IDD, signature scheme names, which identify public key and hashing algorithm pairs, are taken from the TLS IANA standard [8].

Field	Type	Description
"timestamp"	DateTime	Date and time of signature creation.
"scheme"	String	A name defined in [8], such as "ecdsa_secp256r1_sha256".
"sum"	String	Base64 encoded [7] signature sum.

3.7 Canonical Forms

Values conforming to some of the types in this document will have to be hashed to produce necessary identifiers. This requires that all relevant values can be presented in a canonical form. The canonical form of all types described in this document are produced by encoding them in JSON, but with the following restrictions:

1. UTF-8 encoding must be used.
2. No insignificant whitespace may be used, as defined in [3].
3. No String escapes may be used. String are to be treated as raw UTF-8 byte arrays enclosed with double quotes, even if they contain illegal UTF-8 characters.
4. Integer Numbers must
   (a) not have a leading minus sign if zero,
   (b) have no fraction or exponent, and
   (c) must not contain any leading zeroes, unless exactly 0.
5. All other Numbers must
   (a) contain an integral of exactly 1 non-zero digit,
   (b) have a fraction, unless it is effectively 0,
   (c) not have trailing fraction zeroes,
   (d) have an exponent, unless it is effectively 0,
   (e) use capital ”E” as exponent marker,
   (f) not have a leading exponent plus sign,
   (g) not have any trailing exponent zeroes.
6. Object pairs must be provided in the same order as they are listed in their type definitions. If no type definition exists, due to the Object being interpreted as being an arbitrary mapping between keys and value, the pairs must be sorted in ascending alphabetical order by their keys.
7. Object pairs with values of the Null type must be omitted.
8. DateTime types must be represented by strings with the ”YYYY-MM-DDTHH:MM:SS(.s+)Z” form outlined in the DateTime description, with the millisecond part specified with any trailing zeroes omitted. If only zeroes are part of the millisecond part, the part is omitted entirely (including its leading dot).

Note that the Numbers `0`, `-1` and `12500` are canonical integers, while `-0`, `0.0` and `002` are not. Furthermore, the decimal Numbers `5`, `1.025E3` and `4E-9` are canonical, while `7.0`, `12E4`, `7.10`, `4E+9`, `6.2E02` and `3E0` are not. Also note that some integer numbers are represented in exactly the same way as decimal numbers without fractions and exponents.

3.7.1 Examples

The following is an example of a Hash in a valid canonical form.

```json
{"algorithm":"md5","sum":"YFfxPEluz3/Xd86555rihQ=="}
```

The next example, however, is not in a valid canonical form, as the order of the pairs does not correspond to that of the Hash specification in this document.

```json
{"sum":"YFfxPEluz3/Xd86555rihQ==","algorithm":"md5"}
```

Other things that could make a representation non-canonical are whitespaces or incorrectly capitalized pair names, as follows.

```json
{"hashFunction":"md5", "sum":"YFfxPEluz3/Xd86555rihQ=="}
```

The following is the Offer in Listing 2 in canonical form, with the exception that some insignificant whitespace has been added to improve readability. Note that insignificant spaces are removed between object name/value pairs and that the contract arguments are listed in alphabetical order.

```
{
  "negotiationId":55132,
  "offerorFingerprint":{"algorithm":"md5","sum":"r9y2W1WeiI+6qmMkr/Hz3w=="},
  "receiverFingerprint":{"algorithm":"md5","sum":"cmPMoJPtIhUM9tW37sCWvA=="},
  "validAfter":"2020-06-04T11:39:29.26Z",
  "validUntil":"2020-06-05T11:39:29.26Z",
  "contracts":[{
    "templateHash":{"algorithm":"md5","sum":"skfh83djseiI+4qmMZ//xz5W=="},
    "arguments":{
      "articleNumber":"ABC-123",
      "buyer":"A",
      "price":"3750",
      "quantity":"12",
      "seller":"B"
    }
  }],
  "signature": {
    "timestamp":"2020-06-04T11:39:29.26Z",
    "scheme":"rsa_pkcs1_sha1",
    "sum":"+O43eyod3SadCDe3MMpJ+K2BFTA="
  }
}
```

4 References

[1] R. Fielding and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing," RFC 7230, 2018, RFC Editor. [Online]. Available: https://doi.org/10.17487/RFC7230

[2] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, 2018, RFC Editor. [Online]. Available: https://doi.org/10.17487/RFC8446

[3] T. Bray, "The JavaScript Object Notation (JSON) Data Interchange Format," RFC 7159, 2014, RFC Editor. [Online]. Available: https://doi.org/10.17487/RFC7159

[4] M. Cowlishaw, "IEEE Standard for Floating-Point Arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, July 2019. [Online]. Available: https://doi.org/10.1109/IEEESTD.2019.8766229

[5] "Date and Time on the Internet: Timestamps," RFC 3339, 2002, RFC Editor. [Online]. Available: https://doi.org/10.17487/RFC3339

[6] IANA. (2019) Hash Function Textual Names. [Online]. Available: https://www.iana.org/assignments/hash-function-text-names/hash-function-text-names.xhtml

[7] "The Base16, Base32, and Base64 Data Encodings," RFC 4648, 2006, RFC Editor. [Online]. Available: https://doi.org/10.17487/RFC4648

[8] IANA. (2019) Transport Security Layer (TLS) Parameters – TLS SignatureScheme. Note the table *Description* column. [Online]. Available: https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml#tls-signaturescheme

5 Revision History

5.1 Amendments

No.	Date	Version	Subject of Amendments	Author
1				

5.2 Quality Assurance

No.	Date	Version	Approved by
1			