Annals of Telecommunications (2021) 76:119–130
https://doi.org/10.1007/s12243-020-00792-1

Toward unified trust and reputation messaging in ubiquitous systems

David Jelenc[1]

Received: 17 February 2020 / Accepted: 8 July 2020 / Published online: 7 August 2020

Abstract
The fifth mobile generation (5G) will enable massive distributed applications that run on various platforms and cater diverse and interacting entities. If such interactions are to be successful, the entities will have to learn to trust each other and one way of addressing this is to use trust and reputation systems. These systems estimate the trustworthiness of potential interaction partners and are now being increasingly deployed. However, their inability to share information across applications is concerning: as entities traverse application boundaries their trust and reputation information does not. Instead, it is kept in silos forcing entities to remake it in every application they join. The lack of appropriate standards further impedes such sharing attempts. To address this, we propose a general framework for facilitating the exchange of trust and reputation information. The framework defines messages and a protocol that allows trust and reputation systems to query each other for ratings, provide responses, and signal errors. We analyze the proposal and provide an implementation as free software.

Keywords Trust · Reputation · Messaging · Standardization · Interoperability

1 Introduction

With the proliferation of new network technologies, such as 5G, and new computing paradigms, such as the Internet of Things, collaborative applications in which different entities interact (people, devices, services) are becoming increasingly more common. These applications may be running on diverse platforms ranging from powerful hardware in the cloud to power-constrained embedded devices; we herein refer to such systems as ubiquitous. Establishing collaboration between entities in ubiquitous systems is difficult, because entities can only hope that their interaction counterparts will honor agreements. This is even further emphasized in cases when entities have no prior relationship, or in open environments that lack arbiters to settle disputes. To address such cases, trust and reputation systems have been proposed. These systems collect, estimate and disseminate data about entities' trustworthiness which can be later used for making various decisions: with whom to interact in a commercial exchange, who to query for information and similar. And while the difference between trust and reputation is well established, the distinction between trust and reputations systems is often blurred; herein we use the terms trust and reputation systems interchangeably.

Trust and reputation systems vary greatly. Some are used in centralized settings (e.g., electronic marketplaces [1]) and some in distributed (e.g., peer-to-peer networks [2]), some aid the decision-making of humans, and some operate within automated tasks like routing [3]. In spite of these differences, all systems have the same objective: estimate trustworthiness of entities. But as entities traverse applications boundaries, their trust and reputation does not: it is kept in silos and made available only to the application that generated it. If this issue was addressed, we would reap three benefits [4]. First, we would improve the accuracy of trust estimations in existing applications. Second, we would allow entities to leverage trust and reputation across different applications and not require of them to build it from scratch in every application they join. And third, we would accelerate the establishment of trust in new applications (e.g., establishing trust among objects in the Internet of Things systems [5, 6]). The ability to exchange trust and reputation across applications is this considered an open challenge [7].

✉ David Jelenc
david.jelenc@fri.uni-lj.si

1 Faculty of Computer and Information Science, University of Ljubljana, Večna pot 113, 1000 Ljubljana, Slovenia

To exchange trust and reputation we have to address four aspects. First, we have to define messages that carry trust and reputation. The difficulty here lies in their variety: defined messages have to be general enough to cover a broad spectrum of models. Second, we have to define a protocol that exchanges these messages; this includes sending requests and providing responses. Third, we have to estimate the trustworthiness of responses: some sources may not be trustworthy and this should be considered during data integration. And forth, we have to provide translation mechanisms that convert trust and reputation used in one model to the form that can be consumed by the other.

Herein we provide three contributions that address the first and the second aspect. First, we propose a general message structure that supports the most common types of models: those that estimate trust from ratings. We define an abstract rating that represents either an input to, or an output of a trust and reputation model. Second, we design a protocol that facilitates the exchange of such messages. And third, we implement the proposal in Java, Python and C, and offer it as free software. We deliberately omit the aspect of information trustworthiness estimation and information translation. The reason is that the first is typically already covered by the model while the second is profoundly model specific. For instance, estimating the trustworthiness of information is one of the tasks of trust and reputation models: most of them already modulate incoming information based on the trustworthiness of the provider. And if not, models can integrate one of general approaches [8]. Either way the integration should be driven by the model and not by the exchange protocol. Similarly, the information translation should be addressed by administrators of systems that exchange information, since they know best how to convert incoming ratings into their domains. And while the translation can be ad-hoc, there are also general frameworks that re-scale ratings and account for bias [9, 10].

The paper has seven sections. Section 2 defines the basic rating message, Section 3 describes the protocol for their exchange, and Section 4 provides a mechanism that simplifies message creation and parsing. Section 5 analyses the proposal, Section 6 surveys the related work, and Section 7 concludes the paper.

2 Basic trust and reputation message

When discussing messaging between trust and reputation systems, the types of information with which algorithms—trust and reputation models—operate takes the most relevance. This includes both the type of information from which trust is inferred (inputs) as well as the type of information in which trust is conveyed (outputs). Therefore the objective of this section is to define a general message applicable to the majority of trust and reputations models.

2.1 Trust model inputs and outputs

Even when focusing only their inputs and outputs, trust models still vary greatly. On the side of inputs, models consume assessments from previous interactions, termed experiences, and word-of-mouth information given by other entities, termed opinions. But these are not the only options. For instance, [11, 12] list three sources for estimating trust: besides experiences (and opinions), they also list explicit attitudes and behaviors. While explicit attitudes refer to preconceived notions about trust (e.g., how to trust in the absence of information), behaviors denote patterns of interactions.

Many models combine multiple types to improve estimations. On the side of outputs, the complexity is slightly reduced due to a generally recognized notion of trust. Although no definition is universally accepted, [12] nicely generalize that "trust is a measure of confidence that an entity will behave in an expected manner." Models mostly vary based on how this measure is expressed; examples range from probabilistic to qualitative estimates.

Due to such diversity, it seems intractable to accommodate all trust and reputation models. Therefore, we focus on models that on input consume assessments from previous interactions and opinions obtained from other entities, that is, on models based on ratings: either those that were generated by the entity that uses the trust model (experiences), or those that were obtained from third-parties (opinions). According to the literature [7, 13–15], these types are the most common and the need to support their integration is the greatest. Moreover, other sources, unlike ratings, are typically not expected to be exchanged: they are system specific (like roles in a virtual organization) or obtained by analyzing the environment (like social network structure).

2.2 The `Rating` data-type

We start with a `Rating` data-type that can represent either a trust value that was computed by a trust model upon explicit request, or an assessment that was created by an entity after an interaction took place. The challenge is in being sufficiently general. Looking at a few example trust models and considering findings of multiple surveys, common elements begin to emerge. For instance, according to [16], trust has the following properties: subjectivity, dynamicity, asymmetry, incomplete transitivity and context-dependency. Similarly, [12] states that trust is context-specific, dynamic, propagative, non-transitive, composable, subjective, asymmetric, self-reinforcing and event-sensitive. Although most of these properties tell how trust should be computed, they also suggest how a data-type carrying trust values should be defined. In particular, a trust value should exhibit that trust is subjective and asymmetric, dynamic, and context-dependent. Therefore, we define the `Rating` data-type as given in Listing 1; all data-types are given in ASN.1 [17].

The `Rating` is derived from `SEQUENCE` which models an ordered collection of variables of different types. It contains five fields: `source`, `target`, `service`, `date`, and `value`. The `source` represents the (identity of the) entity that issued the rating, and conversely, the `target` represents the entity to whom this rating was assigned. The contents of these fields should uniquely identify the entities: an email address, a URL, or an X.509 distinguished name would be good examples. Therefore, the type of `Entity` is set to be a string. These two fields cover the requirement ratings be subjective and asymmetric. The `service` component denotes the type of service or interaction for which given rating was created. Like `Entity`, `service` is also a string. The `service` allows ratings to be context-dependent. The `date` denotes the timestamp at which the rating was created. The type of this component is `BinaryTime` which is an integer that represents UNIX time. This component allows ratings to be dynamic. The `value` denotes the actual value of the rating. Its type is `ANY` and it represents a value of arbitrary type. Such ambiguity is required to accommodate the diversity of trust models: the value can be a string, an integer, or a real number; it can be a scalar, a vector, or a tuple—it entirely depends on the model. Having `value` of type `ANY` supports all such cases and we include a few examples of such definitions in Section 3.5.

But such generality has a cost: an isolated `Rating` has unclear meaning, lacks parsing instructions and is not a priori or compile-time type-safe. To determine its meaning, we need to know whether the `Rating` is an assessment created by some entity, or a trust value computed by a trust model. This distinction is important because some models share assessments while others share computed trust values. To properly parse the `value` component and to ensure it contains a valid value, we have to know which ASN.1 data-type it encodes; Section 3 describes a protocol that provides such meta-data. Note that this is the only part of the proposed message structure that is not compile-time type-safe: the remainder of the schema is designed in such a way that any message is a valid protocol message while the contents of the `value` require run-time checking to ensure validity.

```
Rating ::= SEQUENCE {
  source  Entity,
  target  Entity,
  service Service,
  date    BinaryTime,
  value   ANY
}
Entity  ::= IA5String
Service ::= IA5String
```

Listing 1 A `Rating` has five components: two entities (one who assigns it and the other to whom the `Rating` is assigned), the type of service, the date of issue, and the value

3 Message exchange

Besides a general `Rating` definition we also require means for exchanging ratings. In particular, we require (i) means for making queries, (ii) means for responding to queries with results or errors, and (iii) means for obtaining `Rating` definitions.

3.1 The protocol

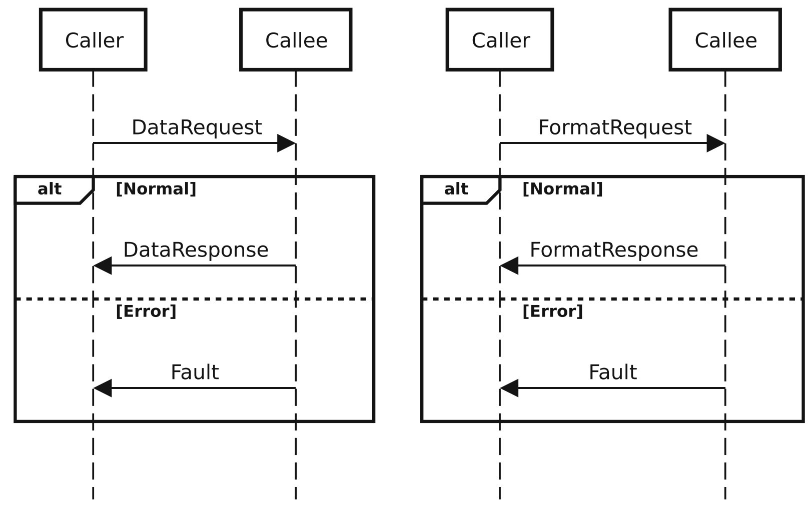
We propose a simple request-response protocol. The protocol is carried out by two peers—trust or reputation systems—that wish to exchange information: one that requests it, herein referred to as the caller, and one that provides it, herein referred to as the callee. We assume a standard peer-to-peer (symmetric) relationship where any peer can enact either role. Each protocol message contains four components: the `version`, the `caller`, the `callee`, and the `payload`. The `version` represents the version of the protocol the peer is using and for all messages in this paper this value is fixed to `1`. Fields `caller` and the `callee` denote the identities of systems that sent and responded to the request respectively. The `payload` determines the semantic of each protocol message: it either contains an instance that requests information or an instance that replies to a pending request. The payload is therefore modeled with the ASN.1 type `CHOICE` which represents a union of one or more alternatives (see Listing 2).

The sequence of the exchange is given in Fig. 1. To request ratings (Fig. 1), the caller sends a `DataRequest`,

```
Message ::= SEQUENCE {
  version INTEGER,
  caller  Entity,
  callee  Entity,
  payload CHOICE {
    data-request     DataRequest,
    data-response    DataResponse,
    format-request   FormatRequest,
    format-response  FormatResponse,
    fault            Fault
  }
}
```

Listing 2 A protocol `Message` contains the `version`, the id of the `caller` and the `callee`, and a `payload` whose content determines the message semantics

Fig. 1 Sequence diagram. Left: a caller requests ratings and the callee responds either with data or with an error. Right: a caller requests `Rating` definitions and the callee responds either with an ASN.1 specification or with an error



(a) Obtaining Rating instances. (b) Obtaining Rating definitions.

to which the callee responds with a `DataResponse` that contains ratings. If an error occurs, the callee instead returns a `Fault`. To request the definitions of `Ratings` (Fig. 1), the caller sends a `FormatRequest` and the callee answers with a `FormatResponse` that contains the definitions in ASN.1 syntax. In case of an error, the callee returns a `Fault`. Within a request-response exchange, the contents of `caller` and `callee` remain unchanged: the first is set to the system that originated the request and the second is set to the system that responded to it. The following sections describe these data-types in more detail.

3.2 Requesting and providing definitions

The definitions of `Ratings`—the definition of the `value` component in each `Rating`—are requested and respectively provided with `FormatRequest` and `FormatResponse` (see Listing 3). A `FormatRequest` is simply an alias for `INTEGER` and denotes a unique message identifier. This number must be chosen anew for every request between the peers.

An answer to a `FormatRequest` is a `FormatResponse` containing five fields. The first is the

```
FormatRequest ::= INTEGER
FormatResponse ::= SEQUENCE {
  rid              INTEGER,
  assessment—id    Format,
  assessment—def   IA5String,
  trust—id         Format,
  trust—def        IA5String
}
Format ::= OBJECT IDENTIFIER
```

Listing 3 A `FormatRequest` requests `Rating` definitions, and a `FormatResponse` returns them

request identifier, `rid`, which must be set to the same value that was provided in the original `FormatRequest`; this links the messages. Components `assessment-def` and `trust-def` are strings that contain ASN.1 definitions of the callee's `Rating` data-type, that is, its `value` component. The first one defines the `value` when providing assessments and the second one defines the `value` when providing trust values. Each definition has a corresponding identifier: `assessment-id` uniquely identifies the definition for assessments and the `trust-id` identifies the definition for trust values. Both fields have type `Format` which is an alias for ASN.1 `OBJECT IDENTIFIER`. These values should be globally unique. Their purpose is to identify provided definitions: if two models use the same definitions, they may use the same identifiers.

3.3 Requesting and providing ratings and signaling errors

The caller requests ratings with a `DataRequest` which has three fields: a request identifier `rid`, a request `type`, and the request `query` (see Listing 4). While the request identifier (a unique number linking a request to a response) and the type (a value denoting whether trust values or assessments are being requested) are straightforward, the `Query` is more involved. We thus explain it separately in Section 3.4. For now, it suffices to say the `query` specifies criteria for ratings.

A `DataRequest` is responded with a `DataResponse` which contains four fields: the `rid` links `DataRequests` with `DataResponses`; the `format` tells which definitions are used in the list of provided ratings; the `type` tells whether the provided ratings represent assessments or trust values; and the `response` contains the list of `Ratings`.

```
DataRequest ::= SEQUENCE {
  rid   INTEGER,
  type  ENUMERATED { trust, assessment },
  query Query
}
DataResponse ::= SEQUENCE {
  rid       INTEGER,
  format    Format,
  type      ENUMERATED { trust, assessment },
  response  SEQUENCE OF Rating
}
Fault ::= SEQUENCE {
  rid     INTEGER,
  message IA5String
}
```

Listing 4 DataRequests request ratings, DataResponses return them, and Faults signal errors

Before we stated the Rating itself does not provide sufficient meta-data: by looking at the Rating alone, we cannot know whether it represents an assessment or a trust value, nor we know how to decode its value. However, if we consider a Rating within a DataResponse and a Message, the missing pieces can be extracted. In particular, the format in DataResponse tells how to parse the Ratings; and the type in DataResponse tells whether the ratings are trust values or assessments. Moreover, based on the callee of the Message and the source of each Rating, the caller knows whether that Rating was created by the callee or by someone else and the callee is relaying it as second-hand information.

If a request (FormatRequest or DataRequest) cannot be successfully answered, the callee can respond with a Fault (see Listing 4). It has two fields: the request identifier, rid, linking it to a FormatRequest (or a DataRequest), and a message containing a human-readable description.

3.4 Filtering requests with queries

Often a caller will want to request only a subset of ratings. To support such cases, we provide a mechanism that allows the caller to specify the query criteria by constraining the domain of Rating components, for instance by stating that it is only interested in ratings within a given time frame. In particular, the caller can constrain the values of source, target, service and date. Additionally, the mechanism supports stating multiple constraints simultaneously. This is implemented with a Query which is either a Constraint or an Expression. The first constrains a single Rating component while the second combines multiple such constraints (see Listing 5).

In the simple case, a Query is a Constraint that constrains Ratings by the values of their components.

```
Query ::= CHOICE {
  con Constraint,
  exp Expression
}
Constraint ::= SEQUENCE {
  operator  ENUMERATED { eq, ne, lt, le, gt, ge }
  value     Value
}
Value ::= CHOICE {
  source  Entity,
  target  Entity,
  date    BinaryTime,
  service Service
}
Expression ::= SEQUENCE {
  operator  ENUMERATED { and, or },
  left    Query,
  right   Query
}
```

Listing 5 A Query defines the scope of requested ratings in a DataRequest. It may constrain a single Rating field with a Constraint or multiple with an Expression

Constraining requires three pieces of information: the Rating component to constrain, a value constant against which the component will be compared, and a binary operator that shall compare the two. The Constraint thus has two components: a value and an operator. The value defines the Rating component and its value while the operator defines the comparison. We provide six operators: eq (equal), ne (not-equal), lt (lower-than), le (lower-than or equal-to), gt (greater-than), and ge (greater-than or equal-to).

If the caller wants to specify multiple constraints, the Query has to be an Expression. The latter represents a logical expression composed of a left and a right operand of type Query, and a logical operator: a conjunction (logical and) or a disjunction (logical or). For instance, an Expression may contain two Constraints or a single Constraint and another Expression that in turn contains additional Constraints.

Although simple, the Query is functionally complete, that is, it supports expressing all possible queries. This is because it has a set of functionally complete operators: conjunction, disjunction and negation. The latter is implicit: a Constraint can be negated by complementing the operator (*equal* with *not-equal*, *grater-than* with *less-than or equal-to*, and *lower-than* with *greater-than or equal-to*) while an Expression can be negated with the De Morgan's laws.[1]

[1]The negation of a conjunction is the disjunction of the negations; and the negation of a disjunction is the conjunction of the negations.

3.5 Example messages

Next we use the ASN.1 value assignment syntax to show a few example messages. Message `formatRequest` in Listing 6 is an example of a `FormatRequest`. Since this is an alias for `INTEGER`, the message simply contains an arbitrary number (here `774`) that represents the request identifier. Message `formatResponse` is an example instantiation of `FormatResponse`. It echoes the incoming `rid`, sets the `assessment-id` and `trust-id` to *some* identifier, and provides the ASN.1 definitions inside `assessment-def` and `trust-def`; the definitions represent a qualitative and ordinal rating scale from [18].

Message `fault` shows how a callee could signal an error.

To request a list of assessments that were created before Apr 26 17:46:40 1970 UTC (`10000000` in UNIX time) and were given to `bob` we would send message `dataRequest` from Listing 7. A response to such a request is given in message `dataResponse`: it echoes back the request identifier in `rid`, provides the `format` and the `type` of `Ratings`, and lists the `Ratings` in the `response`. (Alternatively, in case of an error we would respond with a `Fault`.)

To show the flexibility of the proposed schema, we provide two more example specifications that can be used within a `FormatResponse` message to define either assessments (`assessment-def`) or trust values (`trust-def`); the first is shown in Listing 6 where we had a model [18] that uses qualitative (linguistic and ordered)

```
formatRequest Message ::= {
  version 1,
  caller "alice",
  callee "bob",
  payload format—request 774
}
formatResponse Message ::= {
  version 1,
  caller "alice",
  callee "bob",
  payload format—response: {
    rid              774,
    assessment—id    { 1 1 1 },
    assessment—def   "Formats DEFINITIONS ::=
                     BEGIN
                       QTM ::= ENUMERATED {
                         very—bad, bad, neutral,
                         good, very—good
                       }
                     END",
    trust—id         { 1 1 1 },
    trust—def        "Formats DEFINITIONS ::=
                     BEGIN
                       QTM ::= ENUMERATED {
                         very—bad, bad, neutral,
                         good, very—good
                       }
                     END"
  }
}
fault Message ::= {
  version 1,
  caller "alice",
  callee "bob",
  payload fault: {
    rid 774,
    message "Internal error. Try later."
  }
}
```

Listing 6 Examples of a `FormatRequest`, a corresponding `FormatResponse` and a `Fault` message

```
dataRequest Message ::= {
  version 1,
  caller "alice",
  callee "bob",
  payload data—request: {
    rid 113,
    type assessment,
    query exp: {
      operator and,
      left con: {
        operator lt,
        value date: 10000000
      },
      right con: {
        operator eq,
        value target: "bob"
      }
    }
  }
}
dataResponse Message ::= {
  version 1,
  caller "alice",
  callee "bob",
  payload data—response: {
    rid 113,
    format { 1 1 1 },
    type assessment,
    response {
      {
        source "alice",
        target "bob",
        service "seller",
        date 9000000,
        value '0a0103'H
      },
      {
        source "charlie",
        target "bob",
        service "seller",
        date 8000000,
        value '0a0104'H
      }
    }
  }
}
```

Listing 7 A `DataRequest` for assessments given to `bob` before date `10000000`, and a corresponding `DataResponse`

labels. The new examples are given in Listing 8. The first is from a subjective logic framework [19] where `Rating` values are triples denoting the amount of belief `b`, disbelief `d` and uncertainty `u` regarding trust; each component is a real number from [0, 1] and the components sum to 1. The second example is a comparison-based model [20] where instead of absolute ratings, pairwise comparisons are used to express preferences: the `Rating` denotes how the `other Entity` compares with the `target`: a comparison can be expressed with a `less-than`, an `equal-to`, a `greater-than` and an `incomparable` comparison. We emphasize these examples do not cover all possibilities: as new systems emerge, new `Ratings` will be defined and the proposed framework is flexible enough to accommodate them.

4 Creating and parsing `Query` instances

When specifying complex queries, `Query` instances can become awkward to deal with due to their recursive structure. Consider a criteria that in SQL would be expressed as the following: `target = bob AND (service = seller OR service = buyer) AND date > 20`. The corresponding `Query` instance is given in Listing 9. Evidently, writing such SQL-like statements is easier than constructing recursive data structures. Similarly, parsing larger query instances becomes awkward too. Thus to simplify `Query` manipulations, we provide means for converting string statements into `Query` instances and outline a general strategy for parsing them.

4.1 Creating `Query` instances from string statements

To simplify the creation of queries, we build a string parser using a grammar. The parser converts strings into parse trees which are then turned into `Query` instances.

```
Formats DEFINITIONS ::= BEGIN
SL ::= SEQUENCE {
    b REAL, d REAL, u REAL
}
PWC ::= SEQUENCE {
    other Entity,
    comparison ENUMERATED {
        less−than (0), equal (1),
        greater−than (2), incomparable (3)
    }
}
END
```

Listing 8 Examples of rating definitions. The first represents a rating definition in a subjective logic framework [19], and the second a definition in a system that uses pairwise comparisons instead of absolute ratings [20]

```
message Message ::= {
  version 1,
  caller "alice",
  callee "bob",
  payload data−request {
    rid 591,
    type assessment,
    query exp: {
      operator and,
      left exp: {
        operator and,
        left con: {
          operator eq,
          value target: "bob"
        },
        right exp: {
          operator or,
          left con: {
            operator eq,
            value service: "seller"
          },
          right con: {
            operator eq,
            value service: "buyer"
          }
        }
      },
      right con: {
        operator gt,
        value date: 20
      }
    }
  }
}
```

Listing 9 A more involved `Query`: it requests assessments given to bob for service `seller` or `buyer` since date `20`

We need a simple grammar that supports writing singleton constraints (e.g., `date > 5`), multiple constraints or expressions (e.g., `source = alice AND target = bob`) and parentheses to enforce operator precedence. The grammar is given in Listing 10. It defines a recursive rule `stat` that can take one of four alternatives: it can represent a constraint in the form of `FIELD OP VALUE`; a statement enclosed with parentheses; or two statements combined with a *conjunction* or a *disjunction*. The order of alternatives defines their precedence: constraints before parentheses, parentheses before conjunctions, and conjunctions before disjunctions.

Lexer rules on lines 7–9 define permissible values for tokens `FIELD`, `OP` and `VALUE`. The `WS` token denotes all variants of white space that is ignored.

Using the grammar we build a parser that converts string statements into parse trees. For instance, consider the parse tree from Fig. 2 which we obtain by feeding the parser the following: `target = bob AND (service = seller OR service = buyer) AND date > 20`.

```
grammar Query;
stat:   FIELD OP VALUE  # constraint
    |   '(' stat ')'    # parenthesis
    |   stat 'AND' stat # conjunction
    |   stat 'OR' stat  # disjunction
    ;
FIELD:  'source' | 'target' | 'service' | 'date' ;
OP:     '!=' | '<=' | '>=' | '<' | '>' | '=' ;
VALUE:  [a-zA-Z0-9@\\.]+ ;
WS:     [ \t\r\n]+ -> skip ;
```

Listing 10 A grammar for writing query statements

Finally, we convert parse trees into Query instances. The Query is built during a depth-first tree walk: each node made from a constraint statement becomes a Constraint, and each node made from a conjunction or a disjunction statement becomes an Expression with the appropriately set operator. When visiting nodes that represent parenthesized statements, we create no special messages, but instead continue by recursively processing their children; the purpose of using parenthesis in statements is to enforce the order in which the Query gets built.

4.2 Parsing Query instances

Upon receiving a DataRequest, the callee has to parse it, query its data storage, and then return a DataResponse containing Ratings that conform to the incoming Query. The ratings data storage—a file, a database, or something else—will unlikely be able to consume Query instances directly. Therefore the callee will have to convert the Query into something that is compatible with its data storage. But since that could be anything, creating procedures for every imaginable storage is unfeasible. Instead we outline a general strategy for Query parsing and provide an example procedure that filters Ratings that are kept in memory.

A Query can be parsed recursively. We have to handle two cases depending on whether the Query is a Constraint or an Expression. When processing a Constraint, we test whether a particular Rating component (a source, a target, a service, or a date) satisfies the value constant using the Constraint operator. When processing an Expression, we recursively evaluate the left and right operand and join the results with the Expression operator.

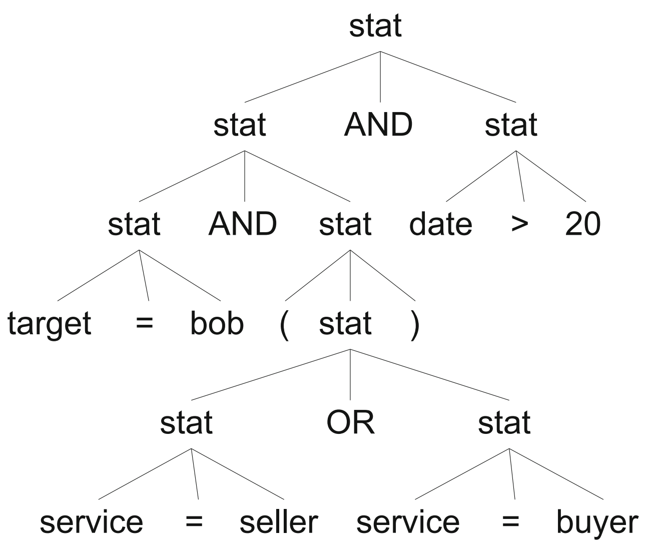


Fig. 2 A parse tree obtained from statement `target = bob AND (service = seller OR service = buyer) AND date > 20` when using grammar from Listing 10

5 Discussion

System architecture The proposal can be applied to centralized and distributed architectures. Centralized systems (e.g., electronic marketplaces) can use it to exchange ratings with similar systems while entities in a distributed system (e.g., nodes of a peer-to-peer network) can use it to exchange ratings with other entities. When a centralized system provides a DataResponse, field callee contains the identity of that system. In a distributed setting, the callee simply contains the identity of the callee entity. System architecture thus determines the peers: in centralized architectures, peers are centralized systems while in distributed architectures peers are entities that comprise the system.

Communication model The proposal is based on a request-response communication model that exchanges data within the requested scope and upon an explicit request. And since messages have identifiers, the exchange can be asynchronous: the caller can send multiple requests to the same callee in parallel and match corresponding responses with identifiers. This allows callees to avoid blocking and respond to requests in any order. Moreover, in asynchronous exchange peers can reuse the communication channel: once established both peers can send requests and receive responses regardless of who initiated it. However, a synchronous exchange is still possible and will likely dominate in centralized systems that have an existing web service infrastructure.

Due to varying capabilities of trust and reputation systems, the proposal is communication channel agnostic. This is to remain applicable to various transports and communication infrastructures: be it RESTful or classic SOA-based web services, direct TCP or UDP sockets, MQTT clients or IPFS nodes in a distributed web. While for the actual information exchange the communication channel (or even the entire communication middleware) is also needed, we expect systems to integrate the messaging systems into their existing infrastructures: since systems are diverse, we cannot require (or enforce) the same communication infrastructure for all; the messages, however, may be the same. For instance, centralized systems might implement this proposal as a REST endpoint that receives a POST request containing a `DataRequest` and responds with a HTTP response carrying a `DataResponse`; the only requirement is that the `content-type` is set to `application/octet-stream` if BER encoding is used. In contrast, low-powered devices, such as nodes in a wireless sensor network, may not have REST capabilities and would use TCP or UDP sockets directly; in those cases asynchronous messaging and the ability to reuse communication channel will be even more important. For the same reasons, the proposal provides no security guarantees. To secure the message exchange, we suggest securing the underlying communication channel.

The proposal and messages are open to modifications while being backward compatible. This is true as long as every breaking change is accompanied by raising the protocol version number. Since all messages start with it, the recipient can parse each message accordingly.

Encoding rules and message size ASN.1 offers multiple encoding rules that range from space-efficient binary representations, such as Basic Encoding Rules (BER), Packed Encoding Rules (PER) and their unaligned version (UPER), to more verbose and human-readable representations such as XML Encoding Rules (XER) and JSON Encoding Rules (JER). These rules determine the sizes of messages as well as the speed with which they are encoded.

To show how the encoding rules affect the message size, we generated a number of messages in which we varied the amount of information, and then encoded them with different encoding rules. We analyzed the sizes of `DataRequests` and `DataResponses`, because they represent the bulk of exchanged information. When making `DataRequests`, we varied the number of constraints: each constraint limited the `source` field to a 16-character string and we combined that constraints with disjunctions. With `DataResponses` we varied the number of `Ratings` in the `response` field. In each `Rating` we set the `source` and `target` to a random 16-character string, the `date` to a random integer, and the `value` to a random instance of a QTM rating [18] (a value form a 5-level scale). Tables 1 and 2 show the message sizes in bytes with respect to the amount of information and the choice of encoding rules.

Compared to BER, `DataRequests` encoded with JER and XER on average consume more space (by a factor of 3.7 and 5.3), while PER and UPER consume less (by a factor of 1.6 and 1.8). Similarly, `DataResponses` encoded with JER and XER on average take more space than BER (by a factor of 2.3 and 3.5), while PER and UPER consume less (by a factor of 1.2 and 1.3).

Encoding rules and speed An important characteristic of any schema is the speed with which messages are encoded and decoded. Here we report on the speeds in Java, C and Python. We are only reporting the speed of BER since these were the only encoding rules available everywhere. We emphasize that the actual speed depends both on the complexity of the schema as well as on the quality of the ASN.1 library that implements it.

When timing encoding, we measured the time needed to convert a message in memory (a struct or a class instance) to its byte representation. When timing decoding, we measured the time needed to convert the bytes that were already loaded into memory to actual messages (instances of structs or classes). We ran all benchmarks on an AMD Ryzen 7 2700X Eight-Core Processor with 16 GB RAM using Ubuntu Linux 18.04 with kernel 5.3.0-59. Python messages were implemented with PyASN1[2] and Python version 3.6.6, Java with jASN1[3] using Java 1.8.0_201, and the C with asn1c[4] and compiled with GCC version 7.5.0. The ASN.1 specifications, its implementations, and the example of a communication channel are available on GitHub.[5]

Results are shown in Fig. 3. For instance, encoding a `DataRequest` with 10 constraints takes 1 µs in Java, 10 µs in C and 424 µs in Python. Encoding 100 constraints takes 11 µs in Java, 86 µs in C and 3927 µs in Python. Decoding a `DataRequest` with 10 constraints takes 1 µs in Java, 18 µs in C and 1221 µs in Python. Decoding 100 constraints Java, C and Python respectively take 14 µs, 162 µs, and 12,091 µs. Coding `DataResponses` is slower than `DataRequests` which is normal since they are bigger. For example, encoding a `DataResponse` with 100 ratings takes 10 µs in Java, 92 µs in C and 3742 µs in Python, while encoding 1000 ratings takes 100 µs, 906 µs, and 41,320 µs respectively. Decoding a `DataResponse` with 100 ratings requires 17 µs in Java, 176 µs in C and 10,311 µs in Python while decoding a `DataRequest` with 1000 ratings takes 168 µs, 1714 µs, and 116,912 µs in Java, C and Python.

While these results are platform dependent, they suggest good performance. Unfortunately, there are no similar benchmarks that would allow us to compare results. Lastly, it looks surprising that a Java-based solution outperforms a C-based one. But as stated, the speeds depend in large on the performance of the ASN.1 library that is used. In our experiments it happens that the Java library was more optimized that the C library.

[2]http://pyasn1.sf.net
[3]https://www.openmuc.org/asn1
[4]http://lionet.info/asn1c
[5]https://github.com/trust-messages.

Table 1 Sizes of `DataRequests` (in bytes) with respect to the encoding rules and the number of constraints

# Constr.	BER	XER	PER	JER	UPER
10	322	1656	211	1138	186
20	602	3171	382	2183	337
30	882	4686	553	3228	488
40	1162	6201	724	4273	640
50	1442	7716	896	5318	791
60	1722	9231	1067	6363	942
70	2002	10,746	1238	7408	1093
80	2282	12,261	1409	8453	1245
90	2562	13,776	1581	9498	1396
100	2842	15,291	1752	10,543	1547

Querying The framework provides querying capabilities allowing the caller to request a subset of ratings by constraining the values of `source`, `target`, `service` and `date`. The mechanism is efficient, self-sufficient and compile-time type-safe. The efficiency comes from the encoding rules; the self-sufficiency from not relying on external systems or tools; and type-safety from the ASN.1 schema which guarantees that a valid `Query` instance is always a valid query—one cannot constrain a non-existing field or use invalid constraint value. However, the querying mechanism does not support constraining `values` of `Ratings`: for instance, we cannot request `Ratings` whose `values` are above certain threshold. Such queries would be possible had we used an ontology and associated languages: Resource Description Framework (RDF) [21] and SPARQL Protocol and RDF Query Language (SPARQL) [22]. However, relying on such external system would somehow lessen the aforementioned benefits: first, since ontologies use the RDF data model, so called subject-predicate-object triples, which are based on text encodings, we would lose efficiency; second, since ontologies are more complex and provide additional functionalities (not all relevant to this use-case), we would grow a rather minimalist solution into a considerable artifact whose applicability in constrained environments is questionable—we would lose self-sufficiency; and third, since ontologies use run-time constraints and schemas, we would lose compile-time type-safety: such mechanism would allow queries that reference non-existing fields or use invalid values and extra run-time checking would be needed. In summary, while a semantic system would be a more capable alternative, it would also bring less efficiency, simplicity and type-safety.

6 Related work

A few works are explicitly dedicated to eliciting commonalities of trust and reputation models. In [23] authors overview several models, extract common properties, and define a functional interface in the form of a pre-standardization recommendation. The interface defines functions that models ought to implement: information gathering, scoring & ranking, entity selection, transaction, and reward & punish. However, the paper does not address information exchange.

Table 2 Sizes of `DataResponses` (in bytes) with respect to the encoding rules and the number of ratings

# Ratings	BER	XER	PER	JER	UPER
100	5361	18,837	4444	12,157	3989
200	10,661	37,437	8845	24,157	7940
300	15,961	56,037	13,245	36,157	11,890
400	21,261	74,637	17,645	48,157	15,840
500	26,561	93,237	22,045	60,157	19,790
600	31,861	111,837	26,445	72,157	23,740
700	37,161	130,437	30,845	84,157	27,690
800	42,461	149,037	35,245	96,157	31,640
900	47,761	167,637	39,645	108,157	35,590
1000	53,061	186,237	44,045	120,157	39,540

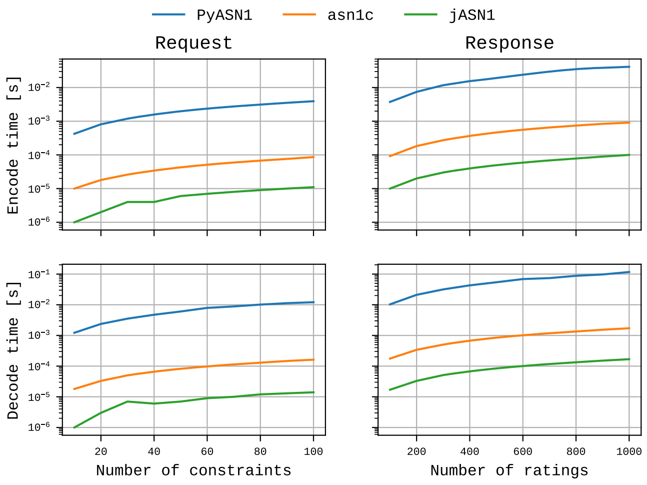


Fig. 3 Encoding speed with respect to the ASN.1 library; `DataRequests` varied in number of constraints and `DataResponses` varied in number of ratings

In [15] several systems are studied and a framework of the desired requirements and features is proposed. The paper addresses aspects pertaining to how trust is expressed and calculated, but not how it should be encoded or exchanged. Similarly, [14] propose a trust and reputation meta-model that defines trust and reputation models and identifies standards in all contexts. The meta-model defines model requirements: scope, goals, fundamental concepts, context and susceptibility to attacks. It requires the definitions be merely conceptual: the issue of defining unified data-types or exchanging them is unaddressed.

One of the early attempts to trust and reputation exchange was given in [24]. They propose an XML DTD that defines two messages, `trustRequest` and `trustResponse`, that facilitate the exchange, but they are tied to a specific trust model. Similarly [25] propose a SOA-based web service to implement an interface to trust and reputation systems. The web service allows agents to look-up trust and reputation of other agents as well as submit ratings. They define a rating data-type and a service interface. But the data-type is tied to a specific model and the proposal lacks querying facilities.

In [26] a general purpose rating ontology is proposed. It has six components and the first four (`about`, `submittedBy`, `creationTime`, `hasAspect`) directly correspond to `target`, `source`, `date` and `service` from Listing 1. The rating value is split in two: `hasScale` and `hasValue`. The first determines the type of the scale (nominal, ordinal, interval, or ratio), and the second determines the actual value. Contrary to this proposal, the ontology can only accommodate scalar ratings; multidimensional values are unsupported. Similarly, an evaluation framework Alpha Testbed [27] also defines ratings as 5-tuples containing `source`, `target`, `service`, `date` and `value`. While the first four correspond to fields from this proposal, the `value` is a scalar from [0, 1].

In [4] a sharing model termed cross-community reputation (CCR) is proposed. The proposal focuses the issue of information translation by converting ratings into universal set of context and values. CCR is extended with an architecture proposal, called Trust and Reputation In virtual Communities (TRIC) [28]. They propose centralized architecture where the TRIC server plays the role of a information exchange proxy. The authors also discuss three aspects of exchanging information: the initiator (whether data is pushed or pulled), the trigger (whether the exchange is on-demand or periodical) and the sensitivity (what kind of data change triggers an exchange). The CCR and TRIC are promising solutions for centralized communities. However, assuming that all models use similar rating definitions and that universally defined contexts exists is somewhat limiting. Nevertheless, neither proposal defines concrete data-types (messages), explicit service interfaces, nor querying mechanisms.

7 Conclusion

We proposed a general rating message that represents either an input to or an output of a trust and reputation model and is flexible enough to accommodate the majority of existing systems. We defined a service interface and a set of related messages that allow trust and reputation systems to query each other for and respond with ratings or signal errors. To simplify message creation, we defined a grammar that allows creating queries from strings and provided a procedure that parses such query messages. We implemented the proposal in Java, C, and Python and released the implementation and message specification as free software.

The proposal is a step closer toward sharing trust and reputation across applications. It allows existing applications to estimate trust more accurately, be it toward existing entities or toward new ones. New applications can use it to speed up the establishment of trust between its members by importing it from other related domains.

Acknowledgments We are grateful to Prof. Denis Trček for invaluable comments and discussions that lead to this work. We also thank Aleks Huč and Balu Deokate for their help in revising the manuscript.

References

1. Post A, Shah V, Mislove A (2011) Bazaar: strengthening user reputations in online marketplaces. In: Proceedings of NSDI'11:

8th USENIX Symposium on Networked Systems Design and Implementation
2. Meng X, Li T, Deng Y (2016) Prefertrust: an ordered preferences-based trust model in peer-to-peer networks. Journal of Systems and Software
3. Khatoun R, Begriche Y, Dromard J, Khoukhi L, Serhrouchni A (2016) A statistical trust system in wireless mesh networks. Annals of Telecommunications
4. Grinshpoun T, Gal-Oz N, Meisels A, Gudes E (2009) Ccr: a model for sharing reputation knowledge across virtual communities. In: IEEE/WIC/ACM International joint conferences on web intelligence and intelligent agent technologies
5. Loper ML, Swenson B (2017) Machine to machine trust in smart cities. In: IEEE 37Th International Conference on Distributed Computing Systems (ICDCS)
6. Yan Z, Zhang P, Vasilakos AV (2014) A survey on trust management for internet of things. Journal of network and computer applications
7. Hendrikx F, Bubendorfer K, Chard R (2015) Reputation systems: a survey and taxonomy. Journal of Parallel and Distributed Computing
8. Parhizkar E, Nikravan MH, Zilles S (2019) Indirect trust is simple to establish. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence. International Joint Conferences on Artificial Intelligence Organization
9. Pinyol I, Sabater-Mir J, Cuni G (2007) How to talk about reputation using a common ontology: from definition to implementation. In: Ninth workshop on trust in agent societies
10. Dondio P, Longo L, Barrett S (2008) A translation mechanism for recommendations. In: Trust Management II - Proceedings of IFIPTM 2008: Joint iTrust and PST Conferences on Privacy, Trust Management and Security
11. Ruan Y, Durresi A (2016) A survey of trust management systems for online social communities–trust modeling, trust inference and attacks. Knowledge-Based Systems
12. Sherchan W, Nepal S, Paris C (2013) A survey of trust in social networks. ACM Computing Surveys (CSUR)
13. Pinyol I, Sabater-Mir J (2011) Computational trust and reputation models for open multi-agent systems: a review. Artificial Intelligence Review
14. Costagliola G, Fuccella V, Pascuccio FA (2014) Towards a trust, reputation and recommendation meta model. Journal of Visual Languages & Computing
15. Vavilis S, Petković M, Zannone N (2014) A reference model for reputation systems. Decision Support Systems
16. Cho JH, Chan K, Adali S (2015) A survey on trust modeling. ACM Computing Surveys (CSUR)
17. ITU-T (2015) Itu-t recommendation x.680: Information technology – abstract syntax notation one (asn.1): specification of basic notation. Technical report, International Telecommunication Union
18. Jelenc D, Trček D (2014) Qualitative trust model with a configurable method to aggregate ordinal data. Autonomous Agents and Multi-Agent Systems
19. Jøsang A (2016) Subjective logic. Springer, Berlin
20. Centeno R, Hermoso R (2018) Estimating global opinions by keeping users from fraud in online review systems. Knowl Inf Syst 55(2):467–491
21. Lanthaler M, Cyganiak R, Wood D (2014) RDF 1.1 concepts and abstract syntax. W3C recommendation W3C. http://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/
22. SPARQL 1.1 overview. W3C recommendation, W3C (2013). http://www.w3.org/TR/2013/REC-sparql11-overview-20130321/
23. Mármol FG, Pérez GM (2010) Towards pre-standardization of trust and reputation models for distributed and heterogeneous systems. Computer Standards & Interfaces
24. Trček D (2004) Towards trust management standardization. Computer Standards & Interfaces
25. Kovač D, Trček D (2009) Qualitative trust modeling in SOA. Journal of Systems Architecture
26. Marienfeld F, Höfig E., Horch A, Kintz M, Finzen J (2011) Making sense of ratings: a common quantitative feedback ontology. In: Proceedings of the 7th International Conference on Semantic Systems
27. Jelenc D, Hermoso R, Sabater-Mir J, Trček D (2013) Decision making matters: A better way to evaluate trust models. Knowledge-Based Systems
28. Gal-Oz N, Grinshpoun T, Gudes E, Friese I (2010) Tric: an infrastructure for trust and reputation across virtual communities. In: Fifth International Conference on Internet and Web Applications and services (ICIW)