Supporting Robust and Secure Interactions in Open Domains through Recovery of Trust Negotiations

Anna Cinzia Squicciarini
Computer Science Department
Purdue University
squiccia@cs.purdue.edu

Alberto Trombetta
Dipartimento di Informatica e Comunicazione
Universitá degli Studi dell' Insubria
alberto.trombetta@uninsubria.it

Elisa Bertino
Computer Science Department
Purdue University
bertino@cs.purdue.edu

Abstract

Trust negotiation supports authentication and access control across multiple security domains by allowing parties to use non-forgeable digital credentials to establish trust. By their nature trust negotiation systems are used in environments that are not always reliable. In particular, it is important not only to protect negotiations against malicious attacks, but also against failures and crashes of the parties or of the communication means. To address the problem of failures and crashes, we propose an efficient and secure recovery mechanism. The mechanism includes two recovery protocols, one for each of the two main negotiation phases. In fact, because of the requirements that both services and credentials have to be protected on the basis of the associated disclosure policies, most approaches distinguish between a phase of disclosure policy evaluation from a phase devoted to actual credentials exchange. We prove that the protocols, besides being efficient, are secure with respect to integrity, and confidentiality and are idempotent. To the best of our knowledge, this is the first effort for achieving robustness and fault tolerance of trust negotiation systems.

1 Introduction

Trust negotiation represents an effective approach to the problem of authentication and access control among multiple security domains by allowing parties to use non-forgeable digital credentials to establish trust [15, 2]. In trust negotiation access to a service provided by a party, referred to as service provider, is authorized based on policies specifying properties, encoded through *credentials*, that the requesting party should satisfy. Disclosure of relevant credentials by the requesting party is in turn governed by policies specifying which credentials the service provider should submit before the requested credentials can be disclosed by the requesting party. Trust is therefore established through mutual exchange of certified digital credentials. A credential can be safely disclosed during a trust negotiation if the associated *disclosure policy* is satisfied. Most approaches organize the trust negotiation process according to two main sequential phases. The first phase, referred to as *policy evaluation phase*, determines the disclosure policies associated with the resources involved in the negotiation. The result of this phase is a credential exchange sequence; such a sequence specifies which credential each party should submit to the other party and which credentials a party should have received from the other party before releasing a given credential. The second phase, referred to as *credential exchange phase*, performs the actual disclosure, according to exchange sequence established by the first phase, of the credentials required to successfully complete the negotiation. In this work, we do not take into account approaches to trust management that are not based on policy evaluation and credential exchange phases, like Hidden Credentials [7], Secret Handshakes or Oblivious signature-Based Envelopes [8]. Though trust negotiation principles and systems have been quite investigated with respect to issues such as privacy, safety and efficiency [13, 15], an open issue is represented by the development of suitable recovery strategies. By their nature trust negotiation systems are used in open environments, often on peer-to-peer infrastructures, that are not always reliable. Because a trust negotiation process may require several rounds to reach its final state, failure of the communication means of the parties implies repeating the same rounds multiple times. Besides the computational issues,

27th International Conference on Distributed Computing Systems (ICDCS'07)
0-7695-2837-3/07 $20.00 © 2007 IEEE

this may jeopardize the security of the peers, as the same sensitive information is to be sent multiple times over a non-always secure network. Additionally, when a party has to verify a credential with certification authorities, failures may result in prohibitively long and costly negotiations. We thus believe that enhancing current trust negotiation approaches with recovery protocols is an important requirement. The goal of the work reported in this paper is to propose such a protocol. Our protocol is based on the use of *savepoint* techniques [3] and *message checkpoint* techniques [4] and it has some important features. The first is that it provides strategies that are tailored for each of the two main phases of the negotiation process and are independent from each other. Therefore, in our approach, it is possible to require recovery to be enabled for both phases, or only for one of the two phases. The second important feature is that our recovery protocol is idempotent; therefore, failures occurring during recovery do not affect the correctness of the recovery process. Finally, it is very efficient and thus can be enforced by peers with limited resources, as shown by our implementation results.

The protocols presented in the paper, although applicable to any trust system, are built on top of a specific trust negotiation system developed by us [2]. Trust-$\mathcal{X}$[2] is a system providing a comprehensive solution to trust negotiation management. Aspects of Trust-$\mathcal{X}$relevant to this paper are presented in Section 2. We then present in Section 3 the recovery protocol for the policy evaluation phase. We present details of the adopted savepoint strategy and the algorithms that are executed for recovering after a crash. In Section 4 we prove several important properties of this recovery protocol, that is, integrity, and idempotency. We also prove complexity results; these results show that our approach is very efficient. In Section 5 we then present the details of the recovery protocol for the credential exchange phase. In Section 6, we illustrate details on the implementation of the protocols, and experimental results. The results clearly show the advantage of introducing recovery mechanisms in trust negotiations, even for relatively short negotiations. We analyze related work in Section 7 and conclude the paper in Section 8.

2 Trust-$\mathcal{X}$: a framework for trust negotiation

Trust-$\mathcal{X}$is a comprehensive framework for trust negotiation, providing both an XML-based language, referred to as $\mathcal{X}$-TNL, and a system architecture. $\mathcal{X}$-TNL supports the specification of digital credentials and disclosure policies, which is the key information exchanged during negotiations. Specifically, digital credentials are assertions describing one or more properties of a given subject, certified by trusted third parties.

Protection needs for the release of a resource are expressed by *disclosure policies*. A resource can be either a service, a credential, or any kind of data that needs to be protected. Disclosure policies regulate the disclosure of a resource by imposing conditions on the credentials the requesting party should possess. We now provide a simplified version of the notational language used in our previous work [2]. We denote a credential as a structured object composed of several items corresponding to its attributes. A credential name and constraints on some of its attribute values (if any), is referred to as *term* $\mathcal{T}(C)$. Example of terms are $IdCard(ZIPCode = 2342)$ or $Passport()$. We express disclosure policies as finite set of logic rules, each of the form: $R \leftarrow \mathcal{T}_1(C), \ldots, \mathcal{T}_k(C)$. Here, R is the target resource for which the policy is specified, and $\mathcal{T}_1(C), \ldots, \mathcal{T}_k(C)$ are the terms corresponding the credentials to be disclosed.

Trust-$\mathcal{X}$ also comprises an architecture for negotiation management, which is symmetric and peer-to-peer. Trust-$\mathcal{X}$ negotiation's main goal is to carry out successful negotiation while protecting as possible credential and policy contents.To this extent, Trust-$\mathcal{X}$ negotiation is organized according to four different phases. Policies are disclosed first during the *policy evaluation phase*, and then only those credentials that are necessary for the negotiation success are disclosed during the *credential exchange phase*. Various strategies can be used for approaching trust negotiations, which may or may not include all the depicted phases. To maximize trust negotiation effectiveness, it is crucial -regardless of the specific strategy adopted - to ensure robustness at least of the policy evaluation and credential exchange phase. Both are fundamental to any trust negotiation. Such approach to negotiation has been widely adopted in several others trust negotiation [11, 15] approaches. Thus, throughout the paper, we focus on recovery protocols for those two phases.

An important component of Trust-$\mathcal{X}$ is the *Negotiation Tree*, a data structure keeping track of the negotiation process. The tree is rooted at the requested resource and it is initialized when the negotiation starts. Then, the tree is dynamically built and is expanded as the phase proceeds. More precisely, a negotiation tree T is a tree in which each node corresponds to a term, and edges correspond to policy rules. Formally, a negotiation tree T

is a tuple $T = \langle \mathcal{N}, \mathcal{R}, \mathcal{E} \rangle$, where $\mathcal{N}$ denotes the set of nodes, $\mathcal{R}$ denotes the root of the tree and $\mathcal{E}$ the set of edges. The order of sibling edges is implied by the policy precondition set of each rule. Furthermore, set $\mathcal{E}$ contains two different kinds of edges: *simple edges* and *multi edges*. A simple edge is used to model policies having only one term on the right side component of the associated rule. By contrast, a multi edge links several simple edges in order to represent policy rules having more than one term on their left side. Nodes belonging to a multi edge are thus considered as a whole during the negotiation. Each node of a negotiation tree can assume two different states: the *deliv* state denoting a delivery resource, that is, a credential/resource ready to be disclosed without further requirements, *open* state, meaning that the credential/resource denoted by the node is not yet ready to be delivered. Nodes are appended to the tree according to the policies exchanged and the dependencies among the different terms. The successful completion of the policy evaluation phase is signaled by a portion of the tree rooted at the requested resource consisting only of nodes with a *deliv* state, that we refer to as *valid view*. When the negotiation tree includes a valid view it is possible to determine a trust sequence for the negotiation. The trust sequence can be built by traversing the view according to a specified order defined by the labelling function associated with the tree.

3 Recovery protocols for the policy evaluation phase

As mentioned, the policy evaluation phase consists of message exchanges between the peers involved in the negotiation, conveying disclosure policies. Each time remote disclosure policies are obtained, the party updates its local negotiation tree accordingly. No third party is involved. Therefore if a peer crashes, it loses the negotiation state. Restarting a negotiation may be costly and not always possible. For instance, this is the case if the involved credentials and/or policies are not locally stored by the peers, and need to be retrieved elsewhere before being exchanged. Or, it may be difficult if the peers are moving objects, and have limited connectivity. The natural approach to avoid such loss is to save in stable storage, either locally or remotely, the state of the process, and periodically update the saved state snapshot while the negotiation progresses. Since the data structure keeping track of the negotiation state is the negotiation tree, performing savepoints in our context implies recording specific portions of such negotiation tree. Other techniques may also be adopted. For instance, the involved credentials may be logged along with the exchanged policies. However, such an approach will result in a number of additional information to record. Further, once the negotiation is started again the peers will have to perform additional operations to rearrange such data in order to retrieve the negotiation state from which restart.

The development of a savepoint technique requires that a number of issues be addressed. A first issue is whether the execution of the savepoints at the two peers should be synchronous or asynchronous. On one hand, a synchronous savepoint protocol makes the recovery simpler; on the other hand, it requires that the peers agree on the frequency of the savepoints, which may also not be easy to reach. This is because the negotiating peers may differ with respect to the network and storage resources they have available. Peers may also have multiple or single open negotiations and either run on desktop computers or mobile devices. When such differences arise, it may not be straightforward to reach an agreement on the savepoint frequency. A second issue is related to the storage strategies for saving the negotiation state. In particular, because the negotiation state is represented by the state of the negotiation tree, one needs to determine storage strategies for such a tree. A first possible approach is to save only the tree's paths that may potentially become trust sequences, that is, valid solutions for the negotiation. However, this solution requires the peers to execute algorithms for composition/decomposition of the tree, thus increasing the computational costs of the savepoints. The second approach is to store the state of the entire negotiation tree each time the savepoint is taken. Such an approach has higher storage costs but it can be computationally very efficient if suitable tree linearization techniques are adopted. Finally, a last crucial issue is related to the security of the protocol executed when recovering from a failure. During recovery from failures peers can be exposed to various forms of attack, like identity theft, phishing, denial of services, replays or man-in-the-middle attacks. These attacks already afflict conventional trust negotiation processes. However, when negotiations are carried out in several sessions more attack points are exposed, especially for attacks that aim at stealing peers' identity and at tampering with negotiation data.

In the remainder of this section we present the technical solutions we have devised to address the above issues.

3.1 Trust negotiation savepoints

Whenever a peer temporarily fails, it loses all information about the exchanged policies. The approach we adopt to prevent such loss is based on savepoints. Generally speaking, savepointing methods are based on periodically saving a global state of the protocol to stable storage. In case of a fault, the protocol is restarted from one of these previously saved states. Savepointing-based methods differ in the way processes are coordinated and on the interpretation of a consistent global state. In our context, peers use stable storage devices to periodically save a snapshot of the state of the process. For simplicity, we assume these devices to be local at the system, or accessible via remote connection. Two types of information are collected. The first is the session key $\mathcal{K}_s$ used during the negotiation for encrypting the exchanged messages.[1] This information is saved the first time a savepoint is executed during the session. The second is the state of the negotiation. The savepoint frequency for the tree may vary according to different factors. Peers may set a time window length or may execute the savepoint each time a policy is received. Other approaches are also possible which we discuss in more details in the concluding section of the paper. Because of the large variety of choices that can be made with respect to the savepoint frequency, we have developed a protocol that does not make any specific assumption on such frequency. A key feature of our approach is that it does not require that the two parties adopt the same savepoint frequency: peers will take savepoints independently, regardless of the mutual dependencies that are introduced because of message exchanges.

In addition to the savepoint frequency, the second important issue is related to the strategies for storing the negotiation state on stable storage. Several strategies can be devised. As we have already mentioned, during the policy evaluation phase, the data structure in charge of keeping track of the negotiation state is the negotiation tree. In our work, we have adopted the well-known object serialization strategies developed in the context of object-oriented programming languages. In general, object serialization is a strategy supporting the encoding of an object and all the objects reachable from it into a stream of bytes. The Java language that we have used for implementing the protocol provides ad hoc classes for object serialization and for the complementary reconstruction of the object graph from the stream. Such a technique is particularly suited for savepointing the structure of the negotiation tree, which is actually composed of nodes linked together according to specific rules. Such a strategy makes it possible to recover the entire data structure, without any information loss. If a peer's main resource limit is represented by the memory, the byte stream obtained by the serialization might be sent to a remote repository. In such a case, whenever a savepoint occurs, in addition to the conventional messages exchanged with the counter peer, messages will have to be sent toward the remote server repository, under a secure connection. By contrast, if resource limit is represented by the network connection, data can be saved locally, and restored if required. In both cases, the computational advantage of the recovery process is significant for any negotiation longer than about three rounds, as proven by our experimental results, discussed in Section 6.

[1] Session keys $\mathcal{K}_s$ are established using a standard authentication session, as for example in SSL.

3.2 The recovery protocol

The purpose of the protocol executed upon a failure is twofold: the first is to authenticate peers and the second is to determine the intermediate state of the negotiation from which to restart. Peers authentication is required in order to verify that the peers are those which claim to be, that is, the peers which were negotiating before the failure. Authentication is assured by the use of the session key $\mathcal{K}_s$, saved at the beginning of the negotiation, for encrypting messages conveying hash values exchanged during the recovery protocol. Because such key is only known to the parties which were negotiating before the failure, only these parties can access the information exchanged during the execution of the recovery protocol. Hash values are used for determining the negotiation tree content to validate. Additionally, to further increase protocol security, a session key for the current recovery session is generated and used to encrypt the messages exchanged as part of the recovery protocol.

The main activity that is executed by the recovery protocol, executed upon a failure, is thus to reconcile the negotiation trees that parties have. The version of negotiation tree that each peer retrieves from stable storage may be different with respect to the version of the other peer because of different savepoint frequencies adopted by the peers. Therefore we need to adopt a reconciliation strategy to merge the two different trees.

The protocol starts by comparing the Merkle value [9] computed on the tree content. If the Merkle values computed by the peers are different, the peers proceed to execute a tree comparison on a hashed version of the tree

nodes. The main advantage of using the Merkle technique is that, if the Merkle tree hash values coincide, synchronization is reached by comparing a single hash value and only one round of messages is required. Under the assumption of peers properly taking savepoints during failure free executions, this is not only the best case scenario, but it is also the most probable one, also assuming that policies do not include references to peers' local environment variables the values of which are not sent to the other peer. The Merkle tree hash function is presented in the following definition. Given a tree node n we use the notation $n(T)$ to denote term nodes, while we use $n(party)$ to denote the party owning node n.

Definition 3.1 [Merkle tree hash function] Let $T = \langle \mathcal{N}, R, \mathcal{E} \rangle$ be a negotiation tree. The Merkle hash value associated with T, denoted as $MhN_T(R)$, is defined as follows:

$$MhN_T(R) = \begin{cases} \text{H(n(T)}||\text{n(party))} \text{ if } n \text{ is a leaf node} \\ \text{H((n(T)}||\text{n(party)}||\text{MhN(child(1,n))}||...||\text{MhN(child(k,n)))} \end{cases}$$

where $'||'$ denotes the concatenation operator, $child(i, n)$ denotes the i-th child of node n and $H(..)$ denotes a cryptographic hash function, e.g. MD5.

By Definition 3.1, the Merkle tree hash value is computed only on nodes content and not on the node state.[2] The reason is that the state of nodes might have been updated during last part of the negotiation after the last savepoint. The fact that node states are different does not necessarily imply that recovery is not possible. Recovery can be executed without considering the differences in state values, as long as the tree content, or at least a view of it, has been validated. The states of nodes will be reconciled once the negotiation is recovered.

The recovery protocol is now discussed in more details. We consider here the case of failures due to crash of one of the peers. In case of both peers crash is analogous, and the first peer reawakened can start the recovery. We call such a peer the $initiator$, since it is the one in charge of activating the recovery procedure. The counter peer is called $receiver$ or receiving peer. First, the initiator peer sends a message to the receiver communicating the height of the tree on which the Merkle tree hash value has been computed. The height of the tree is to be revealed to inform the receiver which tree portion has been considered. Then, a further comparison is executed, if the peers' trees do not coincide. Protocol 1 reports the protocol for Merkle hash values comparison. The case of a successful Merkle value comparison is reported from step 1 to step 5.

[2]We recall that each node has a $state$ field indicating whether the corresponding credential can be disclosed.

Protocol 1 Protocol for Merkle hash values comparison.

Require: $Init$: is the peer initiating the protocol, while Rec is the receiving peer. T_i and T_r are the tree version at the initiator and receiver, respectively. $\mathcal{K}_r$ denotes the new session key while $\mathcal{K}_s$ denotes the saved key.

Ensure: Hash values comparison.

1: $Init$: sends the request $e_{Kr}(e_{\mathcal{K}_s}(Request))$
2: Rec : Decipher request $d_{Kr}(d_{\mathcal{K}_s}(Request))$ and send $e_{Kr}(e_{\mathcal{K}_s}(MhN_{T_r}(R)), TreeHeight)$
3: $Init$: Check $d_{Kr}(d_{\mathcal{K}_s}(MhN_{T_r}(R)))$, compute $MhN_{T_i}(R)$ on T_i of height $TreeHeight$.
{ Matching Hash values}
4: **if** $MhN_{T_r}(R) = MhN_{T_i}(R)$ **then**
5: $Init$: Send $e_{Kr}(Ack)$
6: **else**
7: $Init$: Compute $hash(T_i)$ and send $e_{Kr}(e_{\mathcal{K}_s}(hash(T_i)))$ { $MhN_{T_r}(R) \neq MhN_{T_i}(R)$}
8: Rec: Compute $hash(T_r)$ and compute $View1 = TreeMatch(d_{Kr}(d_{\mathcal{K}_s}(hash(T_i))), hash(T_r))$
9: Rec: Send $e_{Kr}(e_{\mathcal{K}_s}(hash(T_r), View1))$
10: $Init$: $View2 = TreeMatch(d_{Kr}(d_{\mathcal{K}_s}(hash(T_r))), hash(T_i))$
{ Initiator checks for view compliance }
11: **if** $View2 = View1$ **then**
12: $Init$: Send $e_{Kr}(Ack)$
13: **else**
14: $Init$: Send $e_{Kr}(Failure)$
15: **end if**
16: **end if**

Example 1 *Suppose a peer, called peer A, crashes. Suppose that the crash occurs as soon as peer B has stored tree T1 of Figure 1. At the beginning of the recovery protocol, peer A matches the Merkle hash values corresponding to trees T0 and T1. Such a matching fails, due to peers' different frequency in performing savepoints during failure-free execution. Peers interaction is reported in Figure 1.*

If the values computed by the two peers are different, the next phase of the protocol is executed. In such phase, the initiator sends the negotiation tree with all the nodes encoded according to a hash function. The receiver then compares the hash-encoded received tree with its own tree in order to generate a consistent tree. At step 8 of protocol 1, the tree comparison is also executed by Function *TreeMatch()*. Function $TreeMatch()$ pseudo code is not reported for lack of space. It consists of comparing hash values of each node and pruning the node whenever it does not coincide or it does not appear in the received tree. The resulting tree, referred to as *tree view* of the tree, is sent back to the initiator to notify it about the result of the validation process. As reported in step 10, the tree view is thus defined starting from the two negotiation trees, one from the initiator and the other one from the receiver, and represents the reconciled version

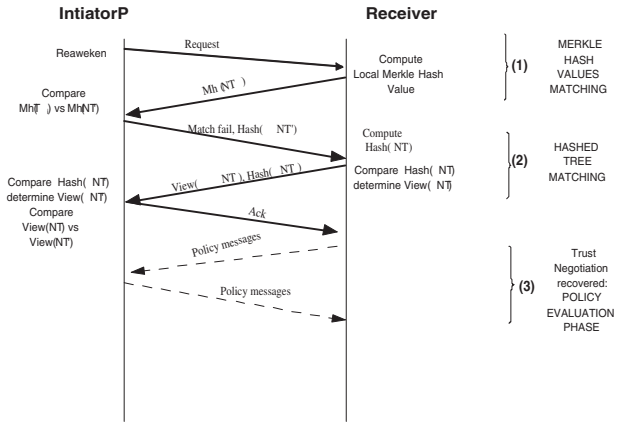

Figure 1. Peers interactions in the negotiation protocol

of these trees. Let T=$\langle \mathcal{N}, R, \mathcal{E} \rangle$ and T'=$\langle \mathcal{N}', R', \mathcal{E}' \rangle$ be negotiation trees. The tree view generated from T and T' is defined as $\mathcal{V}_T = \langle \mathcal{N}'', R'', \mathcal{E}'' \rangle$ where $R = R' = R''$, $\mathcal{N}'' = \mathcal{N}' \cap \mathcal{N}$, and $\mathcal{E}'' = \mathcal{E}' \cap \mathcal{E}$.

Having presented the protocols, we are now ready to introduce the notion of *integrity*, that is specific to our recovery protocol. We say that the recovery protocol verifies integrity if the resumed peers can proceed with the interrupted negotiation using a version of the negotiation tree that does not carry any additional data with respect to that exchanged before the crash. That is, a subtree of the saved negotiation tree of both peers is admitted as reconciled negotiation tree, but no terms and/or policies can be represented in this tree other than those already represented in the trees before the crash. This notion of integrity is formalized through the notion of T-compatibility introduced by the following definition.

Definition 3.2 [*T*-compatibility] Let *initiator* and *receiving* be two negotiating peers and R be a resource. Let $T_i = \langle \mathcal{N}_i, R, \mathcal{E}_i \rangle$ and $T_r = \langle \mathcal{N}_r, R, \mathcal{E}_r \rangle$ be two negotiation trees for R. We say that T_r and T_i are *T*-compatible with respect to a tree T, written $T_i \approx_T T_r$, if one of the following conditions holds:
1. $T = T_i = T_r$;
2. $T = \langle \mathcal{N}', R, \mathcal{E}' \rangle$ is the result of Algorithm *TreeMatch* on input T_i, T_r and $\mathcal{E}' = \mathcal{E}_r \cap \mathcal{E}_i$ and $\mathcal{N}' \neq \emptyset$.

Therefore, whenever T_i and T_r verify the T-compatibility condition with respect to a certain tree T, our integrity notion is verified.

If the protocol succeeds, the subsequent step is to restart the negotiation from the last disclosure policies exchanged. This information is given by the last nodes appended in the tree the peers have agreed on. The negotiation protocol itself suggests which of the peers has to send the next policies in order to proceed.[3]

The case when both peers crash is similar to the case when only one peer crashes. The main difference is that the initiator role can be indifferently played by each peer. The role is actually assigned to the first peer asking to resume the negotiation. In case the peers simultaneously send the request for a recovery, the one which originally triggered the negotiation process is in charge of acting as the initiator.

4 Analysis of the protocols

In this section we analyze the security of Protocol 1 and evaluate its computational complexity. A thorough analysis of such protocol is crucial since it provides the means by which we guarantee that the recovery of the negotiation does not introduce security breaches nor it adds complexity to the system.

4.1 Correctness analysis

We discuss relevant properties of the proposed algorithms, related to integrity and confidentiality. In our context, integrity means that the result of the recovery process does not introduce any data in the negotiation trees that were not already present in the trees before the crash. Confidentiality means – as usual – that no unauthorized third party can gain access to the data exchanged during the recovery protocol. We assume that peers are honest-but-curious, that is, they faithfully follow the protocols and the TreeMatch algorithm steps.

Theorem 4.1 *The recovery protocol satisfies integrity.*

Notice that our protocol does not explicitly deal with protection of data against hacking and tampering. For preventing such threats, we rely on standard encryption techniques.

Confidentiality is guaranteed by the use of encryption. Before starting the recovery protocol, peers agree on a new session key $\mathcal{K}_r$. All the subsequent messages between the negotiating peers during the recovery process

[3] Recall that each level of a negotiation tree always refers to one of the peers, alternatively.

are encrypted with such key and with the old session key $\mathcal{K}_s$. The old session key $\mathcal{K}_s$, besides for confidentiality, is used for authenticating peers, as illustrated in the previous section. Thus, confidentiality relies on the secrecy of the session keys $\mathcal{K}_r$ and $\mathcal{K}_s$. In case confidentiality is violated a third party is unable to access to the message content, since most of the recovery messages will convey hash values, which cannot be deciphered. The worst case scenario happens when the message conveying the view is intercepted by a third party. No actual credentials are anyhow disclosed at this step. As such, the damage is limited to disclosure of non-certified information (i.e., disclosure policies).

A further desirable property for a trust negotiation recovery protocol is to be *idempotent*. Such property is important when dealing with the case of a crash occurring while the peers are executing the recovery protocol. It assures that the recovery protocol can be executed many times and have the same effects as if it were executed only once. Idempotency thus ensures that integrity is achieved despite multiple failures that might occur during the execution of the recovery protocols. We refer to [12] for the proof.

Theorem 4.2 *The recovery protocol verifies the idempotency property.*

4.2 Complexity analysis

We estimate the storage overhead in terms of the number of the nodes of the tree to be saved, whereas we measure the communication complexity in terms of the number of exchanged messages. We also make a comparison between the costs of recovery mechanism and that of resetting the negotiation from the beginning.

Theorem 4.3 *Let T be a negotiation tree containing n nodes. The space complexity required for the recovery procedure is $O(n)$.*

The communication complexity is given in terms of the number of messages and the total size of messages exchanged.

Theorem 4.4 *The worst case communication complexity for the recovery protocol is $O(1)$ in the number of rounds to be performed, and $O(n)$ in the size of the messages, where n is the number of credentials involved in the recovered trust negotiation.*

As such, time complexity of recovery protocol is linear with respect to the number of nodes in the negotiation tree. The complexity of a conventional trust negotiation is usually polynomial in time [14].

5 Recovery protocol for the credential exchange phase

A trivial solution to revcovery from a crash occurred during credentials exchange is to restart the credentials' exchange from the beginning, assuming that the credentials are still valid at the time the exchange is up again. Clearly, this is not a reasonable solution, even more in the resource-bounded setting of mobile environments, in which negotiations may take place. We thus adopt a more effective solution, based on a message-based checkpointing technique [3], according to which the peers' states are piggybacked on the messages the peers exchange. In order to minimize the backup on stable storage of the state of the exchange of credentials, each peer maintains in its volatile memory a list recording every event occurred to the credentials. Whenever a new event occurs (e.g., a credential's request has been received or sent), the peer updates the state and sends it to the other peer, along with the actual message. Whenever a peer temporarily fails, it loses all the information about the exchanged credentials. If the peer reawakens within a fixed time bound, then the other peer sends it its state, along with all its credentials marked as sent and the credential exchange can proceed from the point it was interrupted by the peer's failure.

More precisely, consider two peers P_1 and P_2 that have agreed to exchange credentials $C_1, \ldots, C_n$, as a result of the policy exchange phase. Each peer, for every credential involved in the trust negotiation, keeps track of which events are related with such credential. Possible events are the following: i) a peer can request the credential to the other peer and thus mark such credential as requested, ii) whenever a peer receives the credential's request, the corresponding credential is marked as requested, iii) whenever the requested credential has been sent, such credential is marked as sent, iv) whenever the requested credential is received by the peer, such credential is marked as received. Finally, v) if none of the above events applies, then the credential is marked as not requested. Such information is represented as follows. The *state* of peer P_i is a sequence $S_i = \langle s_1, \ldots, s_n \rangle$, where s_i belongs to the set $\{not\_requested, sent, received, requested\}$ and denotes what is the last event occurred to credential c_i. At the beginning, every item in the state is set to $not\_requested$. A peer can send or receive requests

for credentials, as well as send or receive credentials themselves. According to the event occurred, peer P_i updates accordingly its state S_i:

- If P_i receives a request for credential C_k, then P_i updates s_k to $requested$.
- If P_i sends requested credential C_k, then P_i updates s_k to $sent$.
- If P_i requests credential C_j, then P_i updates s_j to $requested$.
- If P_i receives requested credential C_j, then P_i updates s_j to $received$.

When a peer P_i sends a credential's request or a credential, it sends its current state S_i as well. Then, in the following we assume that peers communicate by messages M composed of requests/credentials and their current states. We now introduce the *SafeSend* and *SafeReceive* Algorithms for sending credentials (and/or their requests, as well) and receiving them, along with the properly updated state. The *SafeReceive* Algorithm executes as follows: once peer P_i receives a message, it updates its state S_i by replacing it with the state $state$ contained in the message. Such state is the current state of the peer that has sent the message. Then, if the content of the rest of the message is a credential, the corresponding entry $S_i.k$ in S_i is updated to $received$. If this is not the case, then the message contains a request for a credential, and the corresponding entry in S_i is updated to $requested$.

The *SafeSend* Algorithm does a similar work in updating the current state of a peer, according to whether the peer has to send a credential or a credential request.

In order to resume the credentials' exchange after a peer's crash, the SafeSend algorithm is however not sufficient yet. The *FaultTolerantSend* algorithm checks, before executing *SafeSend* algorithm (i.e. updating the state and then sending it along with the negotiation's content), that the other peer is active. If this is not the case, the *SafeSend* algorithm periodically checks – within the time limit specified by $timeout$ – whether the other peer has reawakened and in the affirmative case it sends the message M, along with the sent credential lost by the failed peer.

The overhead incurred for sending the peer's state along with ordinary messages (credentials' requests or the credentials themselves) is linear in the number of credential to be exchanged. Furthermore, the execution of *FaultTolerantSend* Algorithm involves sending to the reawakened peer the credentials sent to it, before its crash.

Protocol 2 *FaultTolerantSend* Algorithm

Require: A message M, from peer P_{1-i}
Ensure: Message M to peer P_i, within time bound equal to $timeout$

```
1:  if P_i_is_down then
2:    while timeout do
3:      wait(t);
4:      if (P_i_is_up) then
5:        SafeSend(M);
6:        Send P_{1-i}'s credential marked as sent in S_{1-i};
7:        break
8:      end if
9:    end while
10: end if
```

6 Implementation and experimental results

We have developed the proposed recovery protocols in the context of the Trust-$\mathcal{X}$ system, which wass developed in Java on top of the Oracle database version 10g. Oracle is used to implement a repository storing disclosure policies and the credentials necessary to carry on a trust negotiation. In order to support recovery, the system has been extended with new components supporting the savepoint and recovery of the state of the parties involved in the negotiation process.
We have carried out several tests to evaluate the performance of the extended prototype. Our goal was to identify the threshold, in terms of number of rounds, after which recovery is more efficient than restarting the negotiation from the beginning. The results clearly show the advantage of introducing recovery mechanisms in trust negotiation systems even for relatively short negotiations.

We have performed our experiments on an Intel 1.73GHz processor, with 512MB of RAM using Microsoft Windows XP. The performance of the prototype has been measured in terms of CPU time (in milliseconds). We analyzed the performance of the protocols using two different policy schemas. In the first schema, each peer holds 15 credentials protected by corresponding single term disclosure policies. Based on the experimental results, we determined the threshold, after which recovering is more profitable than restarting the negotiation from the beginning. In the second schema, we used this information to create a tree with the same height as in the previous schema but with an increasing number of nodes after the number of negotiation rounds becomes higher than the threshold.

We carried out two different sets of tests for both phases of our recovery protocols. We present the results of the tests, in which we forced the interruption of the negotia-

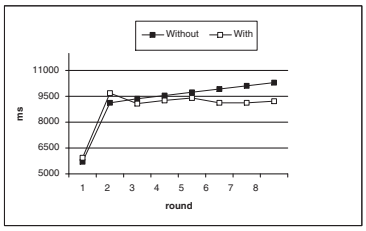

Figure 2. Recovery of the policy evaluation phase

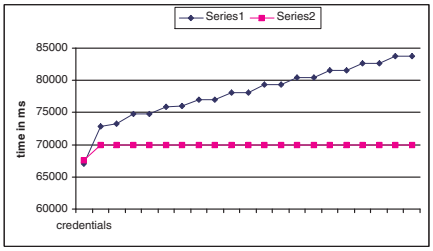

Figure 3. Experiment results for recovery of the credential exchange phase for the first set of experiments

tion at different moments during the tested phase, in order to simulate crashes. We compare the results of the recovered negotiation with the ordinary execution of the same negotiation without the use of recovery protocol, that is, the re-execution to complete the negotiation. The results on the tests for recovery of the policy evaluation phase are reported in Figure 2. In the first set of experiments we observed that negotiations that require at least three rounds register a significant advantage in using recovery with respect to the case in which no recovery protocol is used and thus the negotiation has to start again. As mentioned before, in order to fully assess the behavior of the recovery algorithm, we tested our protocols on a negotiation tree with a more articulated structure, and thus introduced some multi-edges in the portion of the tree to be recovered. We thus updated the policies of the two peers so that the saved levels have several sibling nodes. The results of these experiments confirm that the advantage of recovery increases with the number of nodes for each recovered level of the tree, even for shorter negotiations. We do not report such results here for lack of space. Thus, we conclude that recovery is advantageous if the negotiation has at least two rounds; the advantage is directly proportional to the tree complexity.

The results of the tests on the credential exchange phase are very similar for both set of experiments, because the different organizations of the nodes in the tree do not modify the overall length of the credential path. Figure 3 reports the results of the first set of experiments. Also in these experiments we compared the times for negotiations using the recovery with the times of the same negotiations when no recovery is used. For the case of negotiations without recovery, we report the times required to complete the entire negotiation. For the case of negotiations with recovery, the reported times are obtained by adding: the execution times until the crash; the time for recovery from the last saved checkpoint; the synchronization time (to determine two common checklists); and the time for credential exchange in order to complete the negotiation. As we can see from the graphs, the use of the recovery protocols reduces the execution times, also in the case of short credential sequences.

7 Related work

Trust negotiation for web-based applications has been recognized as an interesting and challenging research area to explore, and it has been the subject of intensive work in the recent years. As a result, a variety of systems and prototypes have been recently developed [11, 13, 15]. In particular, work on trust negotiation has focused on the issue of policy sensitivity. Winslett et al. [15] have designed *Unipro*, a unified scheme to model resource protection, including policies. Seamons et. al [11] introduced policy graphs to safeguard sensitive policies from unauthorized access. A formal framework for trust negotiations has been proposed by Winsborough and Li [13]. They provide an approach for safe enforcement of policies that focuses on a privacy preserving credential exchange.

Concerning system architectures for trust negotiation, Hess et. al. proposed a trust negotiation in TLS (TNT) handshake protocol by adding trust negotiation features [6]. Winslett et al. [15] proposed the TrustBuilder architecture for trust negotiation systems. The TrustBuilder architecture includes a credential verification module, a policy compliance checker, and a negotiation strategy module, which is the core of the system. During a negotiation each agent adopts a local strategy to determine which local resources to disclose and whether to terminate the negotiation. As it is well-known, crash recovery plays a central role both in centralized and distributed database systems and it has been widely investigated [1, 5]. Recently, a lot of work has been done to equip mobile applications with transactional behavior, includ-

ing crash recovery (see [10] for a recent survey on this topic). The recovery algorithms proposed in this work are simplified versions of communication-induced checkpointing protocols, as found for example, in the survey [3]. In particular, we let the peers take independent savepoints without avoiding possibly useless savepoints, as it is usual in more advanced techniques.

8 Concluding remarks

In this paper, we have proposed an efficient and secure approach for recovering from crashes that may occur during trust negotiations. To the best of our knowledge, this is the first time that recovery has been introduced in trust negotiation systems. The implementation and simulation have shown that the communication overheads are low, and definitely motivate the need of such type of protocol in a negotiation system. It is straightforward to adapt the proposed techniques to any trust negotiation system based on tree structure [14]. We will further investigate the applicability of our approach to negotiation systems which do not rely on a negotiation tree structure. Our ultimate goal is to provide a formal foundation for the deployment of reliable trust negotiation systems resilient to unexpected events, like system crash and/or attacks from third parties.

The recovery techniques illustrated in the paper are also a starting point to support a new type of negotiation, referred to as *long lasting* negotiation. Such a negotiation is a trust negotiation carried on in several sessions. Unlike conventional negotiations, in a long lasting negotiation interruptions are intentional and requested by one of the interacting peers. Peers may be willing to temporarily suspend a negotiation if some external or internal events occur.

We will further explore such aspect as part of our future work. Additionally, we will explore how to mitigate certain dangers from a security perspective. The first question we plan to address is how to keep track of each session, and for how long to maintain intermediate states of a negotiation until the session expires. This is crucial to avoid DoS attacks where a peer keeps starting a trust negotiation and does not complete it. The optimal time to track each session in the case many open sessions exist will also be estimated.

Acknowledgments. The work reported in this paper has been partially supported by the National Science Foundation under the ITR Grant No. 0428554 The Design and Use of Digital Identities and by the sponsors of CERIAS.

References

[1] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, 1987.

[2] E. Bertino, E. Ferrari, and A. C. Squicciarini. Trust-χ: A Peer-to-Peer Framework for Trust Establishment. *IEEE Trans. Knowl. Data Eng.*, 16(7):827–842, 2004.

[3] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[4] E. N. Elnozahy and J. S. Plank. Checkpointing for peta-scale systems: A look into the future of practical rollback-recovery. *IEEE Trans. Dependable Sec. Comput.*, 1(2):97–108, 2004.

[5] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[6] A. Hess, J. Jacobson, H. Mills, R. Wamsley, K. E. Seamons, and B. Smith. Advanced client/server authentication in tls. In *NDSS*, 2002.

[7] J. E. Holt, R. W. Bradshaw, K. E. Seamons, and H. Orman. Hidden Credentials. In *WPES '03: Proceedings of the 2003 ACM workshop on Privacy in the electronic society*, pages 1–8, New York, NY, USA, 2003. ACM Press.

[8] J. Li and N. Li. Oacerts: Oblivious attribute certificates. *IEEE Transactions on Dependable and Secure Computing*, 3(4):340–352, 2006.

[9] R. C. Merkle. A certified digital signature. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 218–238, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[10] S. Pleisch and A. Schiper. Approaches to fault-tolerant and transactional mobile agent execution—an algorithmic view. *ACM Comput. Surv.*, 36(3):219–262, 2004.

[11] K. E. Seamons, M. Winslett, and T. Yu. Limiting the disclosure of access control policies during automated trust negotiation. In *NDSS*, 2001.

[12] A. C. Squicciarini, A. Trombetta, and E. Bertino. Supporting robust and secure interactions in open domains through recovery of trust negotiations. *CERIAS Technical Report*.

[13] W. H. Winsborough and N. Li. Safety in automated trust negotiation. In *IEEE Symposium on Security and Privacy*, pages 147–160, 2004.

[14] T. Yu, X. Ma, and M. Winslett. Prunes: an efficient and complete strategy for automated trust negotiation over the internet. In *ACM Conference on Computer and Communications Security*, pages 210–219, 2000.

[15] T. Yu and M. Winslett. A unified scheme for resource protection in automated trust negotiation. In *IEEE Symposium on Security and Privacy*, pages 110–122, 2003.