The Buck Stops Here: Trust Management in Multi-Agent Systems with Accountability

Bilal Khan* Dardo D. Kleiner† David Talmage*

Center for Computational Science
Naval Research Laboratory, Washington D.C.

Abstract

Much of security in multi-agent systems is based on models where each agent declares limits on what other agents are permitted to receive. Traditional systems are engineered to operate without violating their agents' cumulative declared constraints [14, 16].

In contrast, here we consider a trust model that is suited for use by ensembles of closely coupled agents operating in a system supporting agent accountability using audit trails for information flows. In such systems, an agent does not require enforcement of absolute limits on the what other agents receive, but instead seeks assurance that its personal liabilities will never exceed its declared risk tolerance. In short, each agent expects the system to behave in a manner which respects its declared accountability constraints—quantitative limits on what the agent agrees to be held accountable for sending.

This paper outlines a suite of protocols with which a multi-agent system can fulfill the cumulative accountability constraints of its constituent agents, and avoid subjecting any individual agent to greater liability than its declared risk tolerance. The protocols are shown to be efficient in a dynamic network setting, and are analyzed under a comprehensive set of failure models including link delay, link failure, and limited corruption in the control and data processing logic of agents.

1. Introduction

Traditionally, agent security has been approached at the microscopic scale, as a problem of reconciling *pairwise* inter-agent trust with the rendering of agent services [7, 13]. This work, in contrast, considers agent security at macroscopic scales, as a problem of dynamic data filtering in ensembles of cooperating agents [11] within a multi-agent system.

We consider a collection of agents V, cooperating by communicating over a dynamic network of logical connections $E \subset V \times V$. Within the agent network we assume that each datagram p is tagged with an immutable *sensitivity* classification, represented by an m-vector $\bar{\sigma}(p)$ of real values [12]. We shall define a partial order on sensitivity as follows: given two m-vectors $\bar{\sigma}, \bar{\sigma}'$ in $\mathbb{R}^m$, we write $\bar{\sigma} \leqslant \bar{\sigma}'$ if the corresponding ordering holds in *all* m coordinates of $\bar{\sigma}$ and $\bar{\sigma}'$. Larger sensitivity values implicitly mandate greater restrictions on the distribution of a packet.

In multi-agent systems which provide security contracts restricting the content of information flowing *to* individual agents, one interpretation of trust might be to associate with each agent w, the maximum sensitivity of information—say $\tau_1(w)$—that agent w is permitted to *receive* [17]. However, such a model implies that there is global consensus on τ_1—clearly an unreasonable assumption for a dynamic large-scale open multi-agent system. To support interpretation of trust that permits disagreement on trust levels, we could model trust using a pairwise function τ_2, where $\tau_2(v, w)$ specifies the maximum sensitivity of information that agent v wants agent w to *ever receive*. A priori, this model appears to be more flexible, but it is easy to see that a system that satisfies constraints of the second type of model is equivalent to a system based on the first type of model—where for each agent w, $\tau_1(w)$ has been taken to be $\min\{\tau_2(v, w) \mid v \in V\}$. In the dynamic setting then, the difference between the two models amounts to nothing more than maintaining a distributed consensus on the values of τ_1 in terms of the values τ_2. In short, the expressive power of the two trust models is equivalent—no additional flexibility is gained by adopting a pairwise trust function τ_2.

In this paper, we consider a trust model that is suited to open multi-agent systems supporting auditing and accounting of information flows [1, 5, 15].

* Advanced Engineering & Sciences, ITT Industries
† Computer Integration & Programming Solutions, Corp.

At run-time, such a system has two main responsibilities:

(R1) Adhere to quantitative limits specified by each agent on what it is willing to be held accountable for having *sent*.

(R2) Maintaining information regarding the identity of agents complicit in particular information flows.

To make aspect (R1) of the system more precise, we introduce *accountability-based trust*, encoded in a *isolated-accountability function*

$$\tau : V \times V \to \mathbb{R}^m,$$

where $\tau(v, w)$ specifies the maximum sensitivity of information that agent v is willing *to be accountable for having sent* to agent w. Informally, $\tau(v, w)$ represents the declared upper bound on v's personal risk tolerance for sending information to w. In effect, v *declares that it is willing to be held accountable for participating in the routing of information to* w*, provided the information does not have a sensitivity classification higher than* $\tau(v, w)$.

Definition 1.1. *Given an isolated-accountability function* τ*, a system is said to* SATISFY ACCOUNTABILITY-BASED TRUST CONSTRAINTS *if the following is true: Whenever a packet* p *visits a sequence of agents* $w_1, w_2, \ldots, w_l$*, then for all* i, j *where* $1 \leqslant i < j \leqslant l$*:*

$$\bar{\sigma}(p) \leqslant \tau(w_i, w_j).$$

It is important to realize that even though a multi-agent system may SATISFY ACCOUNTABILITY-BASED TRUST CONSTRAINTS, agents are always free to use information collected by aspect (R2) of the system. An agent can use this "audit" information to *subjectively decide* whether "inappropriately sensitive" data is being transferred; the agent can then *autonomously respond* to the circumstances by acting on the the agents complicit in the offending information transfer. While the underlying multi-agent system facilitates subjective decisions and autonomous responses on the part of agents, the system *does not provide any restrictions or guidelines on the decision criteria or the nature of the responses*. Indeed, the optimal decision/response criteria depend strongly on the application domain, and are beyond the scope of this paper. Our goal here is foundational: we describe a suite of protocols which permit a multi-agent system to satisfy accountability-based trust constraints.

2. The Static Case

We are given a static (time-invariant) network of agents modeled as a directed graph $G = (V, E)$, where information flows between agents V along the directed edges E of G. An agent w is called a *parent* of agent v if there is an edge from w to v—equivalently, v is called a *child* of w. The set of all children (resp. parents) of v are denoted $\mathcal{O}_v$ (resp. $\mathcal{I}_v$). Data arrives at an agent v from either "higher layer applications" or from one of its parents. In addition, we are given a static (time-invariant) isolated-accountability function $\tau : V \times V \to \mathbb{R}^m$.

Initialization. Since the network is static, we will assume that every agent knows the structure of G at the onset. From this information, an agent v can compute the set of agents reachable through each of its children. This can be accomplished by running the standard directed depth-first search from each child w in $\mathcal{O}_v$. The results from these depth-first search are to be stored for later reference as values of the *reachability function*:

$$R_v : \mathcal{O}_v \to 2^V.$$

In addition, since the isolated-accountability function is static, we will assume that every agent knows τ at the onset. Each agent computes *contextual-accountability function* τ_v^*, the trust that it places in each of its children within the operational context of the agent network G.

$$\tau_v^*(u) = \min\{\tau(v, w) \mid w \in R_v(u)\}.$$

Since depth-first search take time $O(|V| + |E|)$, the time to for agent v to determine R_v is $O(|\mathcal{O}_v| \cdot (|V| + |E|))$. Having computed R_v, the time required to determine τ_v^* is at most $O(m|\mathcal{O}_v||V|)$. Thus each agent can initialize itself in time $O(m|\mathcal{O}_v||V| + |\mathcal{O}_v||E|)$.

Operation. Suppose v receives a data message p with sensitivity $\bar{\sigma}(p)$. The agent system adds v to p's *chain of custody* field: a list of agents that have processed p. Then v sends p to each $u \in \mathcal{O}_v$ for which $\bar{\sigma}(p) \leqslant \tau_v^*(u)$. An incoming packet can thus be processed in $O(m|\mathcal{O}_v|)$ time. Although initially each agent knows G and τ, after initialization only τ_v^* is needed, which requires at most $O(m|\mathcal{O}_v|)$ space.

Analysis, Part 0. Suppose a packet p visits a sequence of agents $w_1, w_2, \ldots, w_l$. Fix i and j, where $1 \leqslant i < j \leqslant l$. Then w_{i+1} must be in $\mathcal{O}_{w_i}$, and w_j must be in $R_{w_i}(w_{i+1})$. By its definition,

$$\begin{aligned} \tau_{w_i}^*(w_{i+1}) &= \min\{\tau(w_i, z) \mid z \in R_{w_i}(z)\} \\ &\leqslant \tau(w_i, w_j). \end{aligned}$$

Since the control logic of agent w_i decided to send p through w_{i+1}, it must be that $\bar{\sigma}(p) \leqslant \tau_{w_i}^*(w_{i+1})$. So combining, it follows that $\bar{\sigma}(p) \leqslant \tau(w_i, w_j)$. Thus, ACCOUNTABILITY-BASED TRUST CONSTRAINTS (Definition 1.1) are satisfied.

3. Making Isolated-Accountability Dynamic

We now augment the previous description to incorporate a more dynamic setting in which each agent v can alter the

isolated-accountability function τ at domain points of the form (v, w) where $w \in V$. When agent v changes $\tau(v, w)$, it alters its declared risk tolerances for being a party in routing information to w.

Clearly, in such a dynamic setting, it would be unreasonable to assume that every agent knows τ all the time. To address this, the $|V| \times |V|$ table of values encoding the function τ are distributed column-wise across the set of agents V. Each agent v individually maintains a subtable $\tau_v : V \rightarrow \mathbb{R}^m$ where $\tau_v(w) = \tau(v, w)$, and can modify its entries at will. In doing so, it alters its declared risk tolerances for being a party in routing information to other agents.

When an agent v alters $\tau_v(w)$, it must recompute τ_v^* according to the following definition:

$$\tau_v^*(u) = \min\{\tau_v(z) \mid z \in R_v(u)\},$$

for each u in $\mathcal{O}_v$ having the property that $w \in R_v(u)$.

Thus, when agent v changes its isolated-accountability $\tau(v, w)$ to an agent w in V, this can alter its contextual-accountability function. Let us consider the implications of this. Denote the previous value as $\tau_v^*(u)$ and the new value as $\hat{\tau}_v^*(u)$. If $\hat{\tau}_v^*(u) < \tau_v^*(u)$, then v will be more selective about the messages it sends to u, and agents downstream of u may filter fewer messages or even be completely starved of messages. If $\hat{\tau}_v^*(u) > \tau_v^*(u)$, then v will be less selective about the messages it sends to u, and agents downstream of u may have more filtering to do because v sends them more messages. Note that $\hat{\tau}_v^*(u)$ may be incomparable to $\tau_v^*(u)$, in which case no generic statement can be made about the selectiveness with which v filters messages.

Operation. Note that in the dynamic trust case, recomputation of τ_v^* requires us to retain knowledge of R_v, which is computed at initialization time. This requires $O(m|\mathcal{O}_v||V|)$ space. By using a dictionary data structure such as red-black trees, the re-computation of τ_v^* can be achieved in time $O(m|\mathcal{O}_v| \log |V|)$. However, to do this, the trees must be initialized, and the time required to compute initial values τ_v^* becomes $O(m|\mathcal{O}_v||V| \log |V|)$. Hence total initialization time for an agent becomes $O(m|\mathcal{O}_v||V| \log |V| + |\mathcal{O}_v||E|)$.

Analysis, Part 1. The system was previously shown to operate correctly in the case of a static network and a static isolated-accountability function. To see that the above extensions suffice to ensure that a system with dynamic isolated-accountability will satisfy accountability-based trust constraints, note that for any agent v, the contextual-accountability function τ_v^* depends only on v's isolated-accountability τ_v and on the structure of the agent network G (since the latter determines the reachability function R_v for v). In particular, the contextual-accountability function τ_v^* of one agent *does not depend* on the isolated-accountability functions τ_u of other agents $u \neq v$. Thus when an agent v changes its isolated-accountability function, no agent except v needs to update its contextual-accountability function. Since the data path forwards packets based on comparisons between packet sensitivities and contextual-accountabilities, the system will operate correctly provided every agent v updates its contextual-accountability function τ_v^* immediately after it alters its isolated-accountability function τ_v.

4. Making the Agent Network Dynamic

We now extend the previous system to incorporate an even more dynamic setting in which the agent network may change. We would like to allow an agent to request information from additional agents (i.e. expand its set of parents), as well as allowing it to decline receiving information (i.e. contract its set of parents). Specifically, the system will enable each agent v to add and remove parents (edges of the form (w, v) from E). It suffices to address these two operations, since an agent v can then always add and remove children (edges of the form (v, w) in E) by negotiating with them via a separate higher-level protocol.

Initialization. Clearly, in such a dynamic setting, it would be unreasonable to assume that every agent knows G all the time. To address this, the edges in E are distributed across the set of agents V. Each agent v dynamically maintains its incoming and outgoing edges $\mathcal{I}_v$ and $\mathcal{O}_v$ respectively. Initially, we can assume that $E = \emptyset$, so $\mathcal{O}_v = \mathcal{I}_v = \emptyset$ for all agents v, and hence initialization of τ_v^* and R_v can be achieved trivially in $O(1)$ time for each agent.

Operation. The protocol in the dynamic-links case is significantly more complicated. New links can introduce new sets of reachable agents; severed links can make the set of reachable agents smaller. Thus τ_v^* and R_v must be maintained dynamically. In order to handle this dynamism the system separates the *control path* from the data path. The control path is strictly for messages about the structure of the network, and the *control logic* responds to events and messages that describe changes therein. The data path continues to behave as described in the previous sections, passing or filtering messages according to the accountability-based trust constraints.

In the next sections, we will describe the control path protocols in detail. Here we give a very brief summary. When a link (w, v) is added to the network, ADD messages on the control path allow each new agent that can reach v to incrementally adjust its reachability function and recompute its contextual-accountability function. An ADD message bearing the change flows *upward* in the agent network (i.e. *from children to parents*) starting at v. When an agent receives an ADD message from a child, it determines the impact on its reachability function, sends ADD messages

to each parent if needed, waits for acknowledgments from each parent, and then sends an acknowledgment to the child who provided the ADD message. When a link (w, v) is removed from the network, DEL messages flow upward in the agent network, starting at w, allowing each agent that can no longer reach v to incrementally adjust its reachability function and recompute its contextual-accountability function.

4.1. Adding Links

4.1.1. Link Up Events. The LINKUP event self-generated by agents that wish to extend their set of parents. A LINKUP(w) event at agent v indicates that v wishes to be a child of w. When such an event occurs v must inform w about all agents reachable through it. This set is called the *reachable set* of v, and is given by

$$D_v = \bigcup_{u \in \mathcal{O}_v} R_v(u) \cup \{v\}.$$

Because the addition of links takes time (requiring the operation of a network-wide protocol), we must be prepared for several link additions to be occurring in parallel. To accommodate this asynchronous behavior, we use transaction identifiers to tag each link addition. The set of transaction identifiers that identify ongoing transactions at v is denoted $T_v \subseteq V \times \mathbb{N}$.

The agent's actions upon receipt of a LINKUP event are then precisely as follows:

LINKUP(w) at v:

Obtain a new transaction number, $n \in \mathbb{N}$
Create a new transaction identifier, $t \leftarrow (v, n)$
$T_v \leftarrow T_v \cup \{t\}$
Send ADD($D_v, t, 1$) to w

Algorithm 1: LINKUP actions

4.1.2. ADD Messages. The ADD message contains information about a change in the reachable set of an agent. The arrival of ADD($\Delta D, t, k$) message at agent v from agent w signifies that $\Delta D \subseteq V$ is a set of new agents that are now reachable via v because of some LINKUP event (identified by $t = (w_0, n_0) \in V \times \mathbb{N}$ that took place at the agent w_0 which is k network hops away from v.

Agent v determines the change in its own reachable set, $\Delta D \setminus D_v$. If there is no change (or if v has already seen an ADD message with transaction identifier t) then v sends an acknowledgment to w. If, however, there is a change in v's reachable set, v sends an ADD($\Delta D \setminus D_v, t, k+1$) message to each of its parents. Because the protocol is asynchronous, state information must be maintained for forwarded ADD messages until all of them have been fully acknowledged. At each agent, the additional state information includes:

- The child agent which sent agent v an ADD message containing a transaction identifier t is given by $F_v(t)$, and is stored in the map $F_v : T_v \to \mathcal{O}_v$.
- The parent agents to which agent v has forwarded ADD messages containing a transaction identifier t, and that are still unacknowledged, is given by $G_v(t)$, and is stored in the map $G_v : T_v \to 2^{\mathcal{I}_v}$.
- The set of agents specified in an ADD message with a transaction identifier t received by agent v, is given by $D(t)$, and is stored in the map the map $D : T_v \to 2^V$.

The actions upon receipt of an ADD message are then as follows:

ADD($\Delta D, t, k$) at v, from w:

$X \leftarrow \Delta D \setminus D_v$
if $X \neq \varnothing$ **then**
  if $t \notin T_v$ **then**
    $T_v \leftarrow T_v \cup \{t\}$
    $F_v(t) \leftarrow w$
    $D(t) \leftarrow \Delta D$
    $G_v(t) \leftarrow \mathcal{I}_v$
    Send ADD($X, t, k+1$) to every agent $z \in \mathcal{I}_v$
  else
    //`redundant case`
    $R_v(w) \leftarrow R_v(w) \cup \Delta D$
    $\tau_v^*(w) \leftarrow \min\{\tau_v(u) \mid u \in R_v(w)\}$
    $D_v \leftarrow D_v \cup \Delta D$
    Send ACK($t, k-1$) to w
  end
else
  //`useless case`
  Send ACK($t, k-1$) to w
end

Algorithm 2: ADD actions

Analysis, Part 2. An ADD message propagates upward in the network G until it is found to carry information that is either *redundant* (because it has a transaction identifier that has already been seen) or *useless* (it describes a vacuous change in reachable set). At this point the particular branch of the ADD terminates, an ACK message is generated back towards the source of the ADD.

4.1.3. ACK Messages. The arrival of message ACK(t, k) at agent v from agent w indicates that w has received and processed a prior ADD message sent by v to w. The value of k is always the distance (in hops) between v and the originator of t.

When $k = 0$, v is the originator of t and a child of w, so v adds w to $\mathcal{I}_v$. When $k \geqslant 1$, agent v sends ACK($t, k-1$) to $F_v(t)$, the child that sent the initial ADD(_, t, _) message.

ACK(t, k) at v, from w:

$G_v(t) \leftarrow G_v(t) \setminus \{w\}$
if $G_v(t) = \varnothing$ **then**
  if $k = 0$ **then**
    $\mathcal{I}_v \leftarrow \mathcal{I}_v \cup \{w\}$
  else
    if $k \geqslant 1$ **then**
      if $k = 1$ **then**
        $\mathcal{O}_v \leftarrow \mathcal{O}_v \cup \{F_v(t)\}$
        $R_v(F_v(t)) \leftarrow \varnothing$
      end
      $R_v(F_v(t)) \leftarrow R_v(F_v(t)) \cup D(t)$
      $\tau^*_v(F_v(t)) \leftarrow \min\{\tau_v(u) \mid u \in R_v(F_v(t))\}$
      $D_v \leftarrow D_v \cup D(t)$
      Send ACK($t, k-1$) to $F_v(t)$
    end
  end
  $T_v \leftarrow T_v \setminus \{t\}$
end

Algorithm 3: ACK actions

Analysis, Part 3. An ACK message propagates *downward* in the network (i.e. *from parent towards children*), until it reaches the source of the initial ADD, situated at an agent which experienced a LINKUP event. If a LINKUP(v) is experienced at agent w then the only agents affected are those agents u having v in their reachable set D_u. By the earlier Part 2 of the analysis, all of these agents see ADD messages, and ACK messages are generated at the terminal points of the ADD message propagation tree. When agent u forwards an ACK message it certifies that its R_u, D_u, and τ^*_u functions have been suitably adjusted. Moreover, ACK messages are not forwarded until ACKs have been received from all parents. Since w does not get added to $\mathcal{O}_v$ until v has received ACKs from all its parents, it follows that w does not get added to $\mathcal{O}_v$ until every agent u affected by link (v, w), has adjusted its contextual accountability function τ^*_u suitably. It follows that no data can traverse (v, w) until the system can guarantee that accountability-based trust constraints will be satisfied.

4.2. Directed Cycles

The protocol is robust in the presence of cycles. Figure 1 shows a logical network in which v is a child of x, w is a child of v, and x is a child of w. Agent y joins as a child of v. Agent v sends ADD($\{y\}, t$) to x. Agent y is not in D_x and x has not seen t before, so x sends ADD($\{y\}, t$) to w. Agent y is not in D_w and w has not seen t before, so w sends ADD($\{y\}, t$) to w. Agent y changes D_w because w is waiting for an ACK(t) from x but v has seen t before, so it adds y to $R_v(w)$, adjusts the contextual-accountability of w, and sends ACK(t) to w, preventing the infinite loop. The ACK follows the links back from w to x to v. Agent v proceeds according to algorithm 3, adding y to its reachable set and sending ACK($t, 0$) to y, completing the admission of y as a child of v.

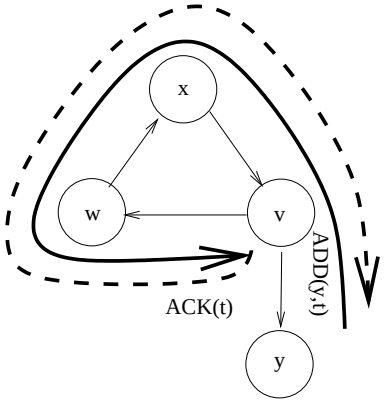

Figure 1. Propagation of ADDs and ACKs in a directed cycle.

4.3. Deleting Links

4.3.1. Link Down Events. The LINKDOWN event is self-generated by agents that wish to contract their set of parents. A LINKDOWN(w) event at agent v indicates that v wishes to be a child of of w no longer. When such an event occurs v must inform w about the agents reachable through v, so that w (and agents further upstream) can adjust their reachable sets and reachability functions appropriately.

The agent's actions upon receipt of a LINKDOWN event are then precisely as follows:

LINKDOWN(w) at v:

Send DEL(D_v, 1) to w
Remove w from $\mathcal{I}_v$

Algorithm 4: LINKDOWN actions

4.3.2. DEL Messages The DEL message contains information about a change in the reachable set of an agent. The arrival of DEL(ΔD,k) message at agent v from agent w signifies that $\Delta D \subseteq V$ is a set of agents that are now no longer reachable via v because of some LINKDOWN event that occurred k hops away in the agent network. In response to a DEL message, agent v must adjust its reachability function by removing the agents specified in ΔD from the $R_v(w)$, and inform its parents about any changes in v's reachable set.

The agent's actions upon receipt of a DEL message are then precisely as follows:

DEL(ΔD,k) at v, from w:
$X \leftarrow \Delta D \setminus \bigcup_{\substack{u \in \mathcal{O}_v \\ u \neq w}} R_v(u) \cup \{v\}$
if $X \neq \varnothing$ **then**
  Send DEL(X, $k+1$) to every agent $z \in \mathcal{I}_v$
end
if $k = 1$ **then**
  $\mathcal{O}_v \leftarrow \mathcal{O}_v \setminus w$
  $X \leftarrow \Delta D \setminus \bigcup_{\substack{u \in \mathcal{O}_v \\ u \neq w}} R_v(u) \cup \{v\}$
else
  $R_v(w) \leftarrow R_v(w) \setminus \Delta D$
  $\tau_v^*(w) \leftarrow \min\{\tau_v(u) \mid u \in R_v(u)\}$
  $D_v \leftarrow \{v\} \cup \bigcup_{u \in \mathcal{O}_v} R_v(u)$
end

Algorithm 5: DEL actions

Analysis, Part 4. A DEL message propagates upward in the network G until it is found to carry information that is *useless* (concerns a vacuous change in reachable set). At this point the particular branch of DEL message propagation terminates. As a DEL message propagates through an agent v, the functions R_v, τ_v, D_v are suitably adjusted so that accountability-based trust constraints will be satisfied.

4.4. Making the Agent Network Open

In the previous section we extended the system to support dynamic links in the agent network. From this point it is straightforward to extend to the case where the set of agents is dynamic. We simply make our agents lenient about the assertion that $T_v \subset V \times \mathbb{N}$, and in doing so, agents no longer need to know the membership of V.

5. Adding Fault Tolerance

We will consider several classes of failures and attempt to make the control-path protocols robust in the presence of faults. First, we shall consider fail-stop crashes of agents and total link failures. For these types of faults, we will use timers to make the control path robust. Then we shall consider more subtle types of failures, namely the corruption of agent data path logic and control path logic.

5.1. Agent and Link Failures

We augment algorithms 2 and 3 to make them support fault tolerance in the presence of fail-stop crashes of agents and total link failures. Additional state is required at each agent, including:

- The set of agents specified in the ADD message with transaction identifier t that was sent to all parents is given by $S_v(t)$, and is stored in the map $S_v : T_v \rightarrow 2^V$.

Under fault conditions, when an agent forwards ADDs to all its parents, it may never receive all the corresponding ACKs. This situation can be detected by starting a timer for each ADD message that the agent processes (see algorithms 6 and 7).

LINKUP-FT(w) at v:
...
Send ADD($D_v, t, 1$) to w
//`fault tolerance`
$S_v(t) \leftarrow D_v$
Start timer(t)
...

Algorithm 6: LINKUP actions modified for fault tolerance

ADD-FT($\Delta D, t, k$) at v, from w:
...
Send ADD($X, t, k+1$) to every agent $z \in \mathcal{I}_v$
//`fault tolerance`
$S_v(t) \leftarrow X$
Start timer(t)
...

Algorithm 7: ADD actions modified for fault tolerance

A timeout occurs at agent v if all ACKs are not received by the timer's expiration. This indicates that some ancestor is taking too long to acknowledge an ADD message or that there is long delay or failure on an upstream link. The protocol recovers (See algorithm 8) by sending a DEL message to each parent ($\mathcal{I}_v$) to remove the agents of the corresponding ADD($S_v(t)$) and by forgetting about t.

The agent's actions upon receipt of a TIMEOUT event are then precisely as follows:

TIMEOUT(t) at v
Send DEL($S_v(t)$) to every agent $z \in \mathcal{I}_v$
$T_v \leftarrow T_v \setminus \{t\}$

Algorithm 8: TIMEOUT actions

Because ACKs may be on their way and cross paths with the DEL messages being used to undo the ADD, the ACK handling algorithm must be prepared to ignore late acknowledgments (see algorithm 9). Removing t from T_v in algorithm 8 prevents the protocol from being fooled by such phantom ACKs.

ACK-FT(t, k) at v, from w:

```
//fault tolerance
if t ∈ T_v then
    ...
    if G_v(t) = ∅ then
        //fault tolerance
        Stop timer(t)
        ...
    end
end
```

Algorithm 9: ACK actions modified for fault tolerance

5.1.1. Phantom ACK. Figure 2 (diagrams 1-5) illustrate the interesting *phantom ACK* problem. They show how the protocol is robust when two slow links delay the delivery of ADD and ACK messages.

In diagram 1, w becomes a child of v. Agent v sends ADD($\{w\}, t$) to its parents using the slow link on the left of the figure and the fast link on the right. The ADD on the fast link arrives at x and x sends ADD($\{w\}, t$) to its parent on another slow link. In diagram 2, x times out waiting for the reply from its parent and sends DEL($\{w\}$) to the parent. It does not know that there is an ACK(t) in transit from one of its ancestors. In diagram 3, the ADD($\{w\}, t$) arrives at x on the slow link. Agent x treats it as a new transaction because it does not remember t. Agent x sends the ADD to its parent on the other slow link. The ACK(t) from its ancestor is still in transit. In diagram 4, the ACK(t) arrives at x. Agent x believes that its ADD succeeded, adjusts the contextual-accountability of the link, and sends the ACK along. There is no way for x to know that the ACK is for the ADD that timed out because both ADD messages are part of the same transaction and thus carry the same t. Eventually, the ACK arrives at v. The second ACK that v expects will never arrive, so v will time out and send DEL($\{w\}$) on both the slow and the fast links as shown in diagram 5. Agent w will time out and detach itself from v.

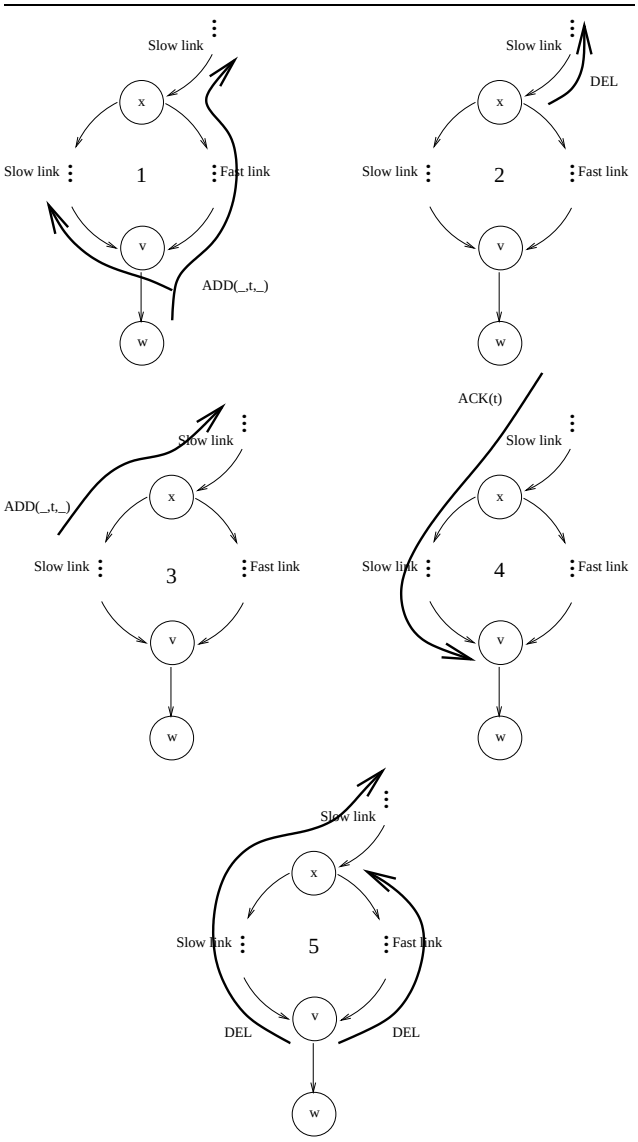

Figure 2. The phantom ACK phenomenon.

5.2. Agent Corruption

We consider first the effect of corruption of data path logic. Such corruption results in either *feast* or *famine* conditions. Feast conditions are said to exist if agent v sends a message p to some $u \in \mathcal{O}_v$ when $\bar{\sigma}(p) > \tau_v^*(u)$. In doing so, agent v acts in violation of its own risk tolerance, making itself vulnerable to liabilities which exceed its isolated-accountability function. It follows that there is no incentive for an agent to corrupt its data path logic on its own initiative to induce feast conditions. Under famine conditions, v does not send p to some $u \in \mathcal{O}_v$ for which $\bar{\sigma}(p) \leqslant \tau_v^*(u)$. In this case, agent v starves agents downstream of u, but this gives rise to a performance issue rather than a violation of the accountability-based trust constraints.

We now turn to corruption of agent control path logic. In this exposition, we divide these consequences into three broad classes: artificially low contextual-accountability, denial of service, and hidden agents. Of these, only the hidden agents phenomenon results in a violation of accountability-based trust constraints.

5.2.1. Artificially low contextual-accountability refers to the situation when agents are being overly conservative, transmitting information that is less sensitive than it could be. Artificially low contextual-accountability is principally a performance issue, because although contextual-accountability may be low, the accountability-based trust constraints of Definition 1.1 are still satisfied. That is, no agent is being subjected to greater liability than its risk tol-

erance as indicated by its isolated-accountability function.

Contextual-accountability may become artificially low when an agent v:

1. **Sends an ADD message with a ΔD that contains $u \notin D_v$ to an ancestor w whose D_w does not contain contain** u. Consider v with no children attempting to become a child of w. Agent v sends ADD($\{v, u\}$) to w. If w has no other routes to u, then $\{v, u\}$ changes D_w and w sets the contextual-accountability of v to $\min(\tau_w(v), \tau_w(u))$. Contextual-accountability will be artificially low if $\tau_w(u) < \tau_w(v)$.
2. **Fails to send a DEL message in response to a TIME-OUT event.** Suppose v is a child of w and u desires to be a child of v. v sends ADD($\{u\}$) to w and w takes a long time to respond. If v times out and does not send DEL($\{u\}$) to w, then should w ever receive the ADD message, it will include $\tau_w(u)$ in its calculation of the contextual-accountability of v. Contextual-accountability will be artificially low if $\tau_w(u)$ is less than all of the other contextual-accountabilities along the path from w to v.
3. **Incorrectly omits agents from ΔD in a DEL message.** This could be because the agent ignores a LINKDOWN event (equivalent to sending DEL($\{\}$) to its parents) or if the agent simply removes some agents or incorrectly calculates the set membership. Suppose there are four agents v, w, x, and y and that they are connected in a graph with edges (y, x), (x, w), and (w, v). If w sends DEL($\{w, v\}$) to x but x sends DEL($\{w\}$) to y, then y will include $\tau_y(v)$ in its calculation of the contextual-accountability x, setting it too low if $\tau_y(v)$ is the smallest.

5.2.2. Denial of Service in the context of the accountability protocol means that an agent is not granted admission onto one or more paths in the agent network. Denial of service occurs when an agent

1. Ignores LINKUP events, or
2. Ignores ADD messages.

When an agent ignores LINKUP events, it denies itself the opportunity to join the agent network and participate in the accountability protocol. When an agent ignores ADD messages, it denies service to descendants. Neither phenomenon results in a violation of the accountability-based trust constraints.

5.2.3. Hidden Agents is a phenomenon which occurs when some agent exists in the network without the knowledge of all agents that should know about it. Hidden agents can make the contextual-accountability functions artificially high. All of the following hidden agent examples refer to Figure 3.

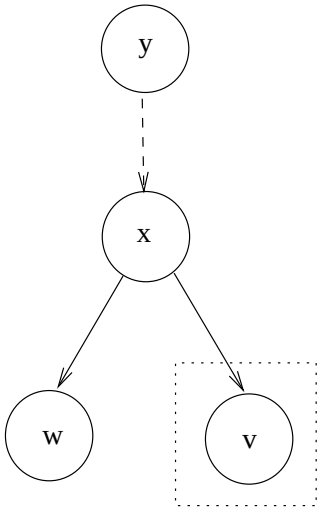

Figure 3. A network with hidden agents.

An agent v can become hidden if:

1. v **is incorrectly omitted from ΔD in an ADD message.** Suppose that agent x with children w and v desires to join agent y. Agent x sends ADD($\{x, w\}$) to y, who has no idea that x is not telling the whole truth. Now v is hidden from y.
2. v **is incorrectly included in ΔD in a DEL message.** Suppose that x is a child of y and that x has children w and v which are known to y. Agent x sends DEL($\{w, v\}$) to y but does not drop the link (x, v). Agent y has no way to know that x did not fulfill its commitment. Agent v is now hidden from y.
3. **another agent sends an ACK instead of forwarding an ADD message.** Suppose that x is a child of y and that x has one child, w, which is known to y. Agent v desires to be a child of x and sends it ADD($\{v\}$). Since this would change x's reachable set, x is supposed to forward the ADD to y. If x instead sends an ACK to v, then v becomes a child of x without the knowledge of y. Thus, v is hidden from y.

Since the admission of a new agent to the network can require lowering the contextual-accountability functions, hidden agents compromise the data path by permitting the transmission of data that is more sensitive than agent risk tolerances permit. There is no way for upstream agents to learn of the compromise.

6. Conclusion

In this paper, we introduced an accountability-based trust model, and defined the criteria for a system to satisfy a set of constraints within this model. We showed that there

is a procedure to for agents to adjust their contextual-accountabilities, even in the presence of dynamic local-accountabilities and a dynamic agent network topology. Finally, we augmented the protocols to withstand fail-stop crashes of agents and total failures of links. We gave a synopsis of possible phenomena that can occur in situations of agent corruption.

Our present software development efforts [8, 9, 10] extend earlier initiatives seeking to harness mobile agents for network management [3, 4, 6] and general information management [2]. In our mobile agent framework, we incrementally construct networks of agents by dragging them from a palette onto a canvas and connecting agents together into a directed graph. We want the agents to decide the sensitivity level of information that flows from producers to consumers and we want them to adjust their contextual-accountability functions so that the system as a whole satisfies accountability-based trust constraints. The protocols described here have been successfully used in the context of our software.

References

[1] K. Aberer and Z. Despotovic. Managing trust in a peer-2-peer information system. In CIKM, pages 310–317, 2001.

[2] Y. Arens, C. Y. Chee, C.-N. Hsu, and C. A. Knoblock. Retrieving and integrating data from multiple information sources. International Journal of Cooperative Information Systems, 2(2):127–158, 1993.

[3] A. Bieszczad, S. K. Raza, B. Pagurek, and T. White. Agent-based schemes for plug-and-play network components. In Proceedings of the 3rd International Workshop on Agents in Telecommunications Applications IATA'98, AgentWorld'98, Paris, France, 1998.

[4] M. M. Cheikhrouhou, P. Conti, and J. Labetoulle. Intelligent agents in network management, a state-of-the-art. Networking and Information Systems, 1(1):9–38, 1998.

[5] Z. Despotovic, K. Aberer, and M. Hauswirth. Trust-aware cooperation.

[6] C. Frei and B. Faltings. A dynamic hierarchy of intelligent agents for network management. In Workshop on Artifi cial Intelligence in Distributed Information Networks (IJCAI'97), 1997.

[7] R. Grimm and B. N. Bershad. Providing policy neutral and transparent access control in extensible systems. Lecture Notes in Computer Science, 1603:311–338, 1999.

[8] B. Khan, D. D. Kleiner, and D. Talmage. CHIME: The Cellular Hierarchy Information Modeling Environment. In Proceedings of International Conference on Parallel and Distributed Computing and Systems 2000, Las Vegas, Nevada, 2000.

[9] B. Khan, D. D. Kleiner, and D. Talmage. Optiprism: A distributed hierarchical network management system for all-optical networks. In Proceedings of IEEE GLOBECOMM, San Antonio, Texas, 2001.

[10] B. Khan, D. D. Kleiner, and D. Talmage. The mother of all databases: A case study in universal situational awareness. In Submitted to IEEE MILCOM, Monterey, California, 2004.

[11] V. R. Lesser. Cooperative multiagent systems: A personal view of the state of the art. Knowledge and Data Engineering, 11(1):133–142, 1999.

[12] G. Pernul, A. M. Tjoa, and W. Winiwarter. Modelling data secrecy and integrity. Data Knowledge Engineering, 26(3):291–308, 1998.

[13] V. Roth. Mutual protection of cooperating agents. Lecture Notes in Computer Science, 1603:275–288, 1999.

[14] V. Swarup and J. T. Fbrega. Trust: Benefi ts, models, and mechanisms. Lecture Notes in Computer Science, 1603:3–18, 1999.

[15] G. Vigna. Protecting mobile agents through tracing. In Third Workshop on Mobile Object Systems, 1997.

[16] J. Vitek and C. J. (Eds.). Secure internet programming - security issues for distributed and mobile objects. Lecture Notes in Computer Science, 1603, 1999.

[17] U. G. Wilhelm, L. Buttyán, and S. Staamann. On the problem of trust in mobile agent systems. In Symposium on Network and Distributed System Security. Internet Society, 1998.