BRICS

Basic Research in Computer Science

Distributed Approximation of Fixed-Points in Trust Structures

Karl Krukow
Andrew Twigg

BRICS Report Series RS-04-16
ISSN 0909-0878 September 2004

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark
Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

`http://www.brics.dk`
`ftp://ftp.brics.dk`
This document in subdirectory `RS/04/16/`

Distributed Approximation of Fixed-Points in Trust Structures

Karl Krukow*† Andrew Twigg‡

September 2004

Abstract

Recently, developments of sophisticated formal models of trust in large distributed environments, incorporate aspects of partial information, important e.g. in global-computing scenarios. Specifically, the framework based on the notion of trust structures, introduced by Carbone, Nielsen and Sassone, deals well with the aspect of partial information. The framework is "denotational" in the sense of giving meaning to the global trust-state as a unique, abstract mathematical object (the least fixed-point of a continuous function). We complement the denotational framework with "operational" techniques, addressing the practically important problem of approximating and computing the semantic objects. We show how to derive from the setting of the framework, a situation in which one may apply a well-established distributed algorithm, due to Bertsekas, in order to solve the problem of computation and approximation of least fixed-points of continuous functions on cpos. We introduce mild assumptions about trust structures, enabling us to derive two theoretically simple, but highly useful propositions (and their duals), which form the basis for efficient protocols for sound approximation of the least fixed-point. Finally, we give dynamic algorithms for safe reuse of information between computations, in face of dynamic trust-policy updates.

1 Introduction

This paper completes a new model for trust in large distributed systems. The model, introduced in papers by Carbone *et al.* [6] and Nielsen *et al.* [14], is

*Supported by SECURE: Secure Environments for Collaboration among Ubiquitous Roaming Entities, EU FET-GC IST-2001-32486

†Basic Research in Computer Science (`www.brics.dk`),
funded by the Danish National Research Foundation.

‡Computer Laboratory, Cambridge University, 15 JJ Thomson Avenue, Cambridge UK

aimed at global-computing environments, and is based on a domain-theoretic modelling of trust information. More specifically, domain theory is applied to give a denotational semantics to a collection of mutually referring so-called trust policies. This provides for a set of principal identities $\mathcal{P}$, each defining such a trust policy, a unique *global trust-state*, which quantifies for any p and q in $\mathcal{P}$, p's trust in q.

Formally, in the framework, trust is something which exists between *pairs* of principals. Each instance of the framework defines a so-called *trust structure*, which consists of a set X of *trust values*, together with two partial orderings of X, the trust ordering ($\preceq$) and the information ordering ($\sqsubseteq$). The elements $s, t \in X$ express the levels of trust that are relevant for a particular application, e.g. access-rights, and $s \preceq t$ then means that t denotes at least as high a trust-level as s. In contrast, the information ordering introduces a notion of precision or information. The key idea is that the elements of the set embody various degrees of uncertainty, and $s \sqsubseteq t$ reflects that t is more precise or contains more information than s. In simple cases the trust values are just symbolic, e.g. $\mathsf{unknown} \sqsubseteq \mathsf{low} \preceq \mathsf{high}$, but they may also have more internal structure. As a simple example of a trust structure, consider the so-called "MN" trust-structure T_{MN} [13]. In this structure, trust values are pairs (m, n) of natural numbers[1], where m denotes the number of "good" interactions and n the number of "bad" interactions. The information-ordering is given by: $(m, n) \sqsubseteq (m', n')$ only if one can obtain (m', n') from (m, n) by adding zero or more good interactions or zero or more bad interactions, i.e. iff $m \leq m'$ and $n \leq n'$. In contrast, the trust ordering is given by: $(m, n) \preceq (m', n')$ only if $m \leq m'$ and $n \geq n'$. Nielsen and Krukow [13, 15], as well as Carbone *et al.* [6], have considered several additional examples.

Given a fixed trust structure $T = (X, \preceq, \sqsubseteq)$, a global trust-state of the system can be described mathematically, simply as a function $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to X$, with the interpretation that $\mathsf{gts}(p)(q)$ denotes p's trust in q, formalised as an element of X. In order to uniquely define the gts function, an approach similar to that of Weeks is adopted [17]. Each principal $p \in \mathcal{P}$ defines a trust policy ("license" in the framework of Weeks), which is an information-continuous function π_p of type $(\mathcal{P} \to \mathcal{P} \to X) \to (\mathcal{P} \to X)$. This function then determines p's trust values, i.e. $\mathsf{gts}(p)$, in the following way. In the simplest case, π_p could be a constant function, ignoring its first argument $m : \mathcal{P} \to \mathcal{P} \to X$. As an example, $\pi_p(m) = \lambda q.t_0$ for some $t_0 \in X$, defines p's trust in any $q \in \mathcal{P}$, as the value t_0. In the general case, π_p may refer to other policies π_z, $z \in \mathcal{P}$, and the general interpretation of π_p is: given that all principals assign trust values as specified in the trust state, $m : \mathcal{P} \to \mathcal{P} \to X$, then p assigns trust values as specified in function $\pi_p(m) : \mathcal{P} \to X$. For example, function $\pi_p(m) = \lambda q \in \mathcal{P}.(m(A)(q) \vee_{\preceq} m(B)(q)) \wedge_{\preceq} \mathsf{medium}$,

[1] To be precise, the set $\mathbb{N}^2$ is completed by allowing also value ∞ as "m" or "n" or both.

represents a policy saying "for any $q \in \mathcal{P}$, give the least upper-bound[2] in $(X, \preceq)$ of what A and B say, but not more than the constant $\mathsf{medium} \in X$".

The collection of all trust policies, $\Pi = (\pi_p : p \in \mathcal{P})$, thus "spins a global web-of-trust" in which the trust policies mutually refer to each other, similarly to a set of mutually recursive functions. A crucial requirement is that the information ordering makes $(X, \sqsubseteq)$ a cpo with bottom. Since all policies are information-continuous, there exists a unique information-continuous function $\Pi_\lambda = \langle \pi_p : p \in \mathcal{P} \rangle$, of type $(\mathcal{P} \to \mathcal{P} \to X) \to \mathcal{P} \to \mathcal{P} \to X$ with the property that $\mathtt{Proj}_p \circ \Pi_\lambda = \pi_p$ for all $p \in \mathcal{P}$, where $\mathtt{Proj}_p$ is the p'th projection[3]. This means that for any collection of trust policies, Π, we can define the unique global trust-state induced by that collection, as the *least fixed-point* of the function Π_λ, denoted $\mathsf{gts}(\Pi) = \mathtt{lfp}\ \Pi_\lambda : \mathcal{P} \to \mathcal{P} \to X$. This unique trust-state thus satisfies the fixed-point equation: for all $p \in \mathcal{P}$

$$\begin{aligned} \mathsf{gts}(\Pi)(p) &= \Pi_\lambda(\mathsf{gts}(\Pi))(p) \\ &= (\mathtt{Proj}_p \circ \Pi_\lambda)(\mathsf{gts}(\Pi)) \\ &= \pi_p(\mathsf{gts}(\Pi)) \end{aligned}$$

Reading this from the left to the right, any function $m : \mathcal{P} \to \mathcal{P} \to X$ satisfying this equation is *consistent* with Π. Consider now two mutually referring functions π_p and π_q, given by $\pi_p = \lambda m.\mathtt{Proj}_q(m)$, and $\pi_q = \lambda m.\mathtt{Proj}_p(m)$. Intuitively, there is no information present in these functions, they simply mutually refer to each other. Thus, we would like the global trust-state induced by these function to take the value $\bot_\sqsubseteq$ on any entry $z \in \mathcal{P}$ for both of p and q. This is exactly what is obtained by choosing the *least* fixed-point of Π_λ.

The trust-structure framework can express many interesting examples, but one could argue against its usefulness as a basis for constructing concrete global-computing trust-management-systems. In order to make security decisions, each principal p will need to reason about its trust in others, that is, the values of $\mathsf{gts}(p)$. When the cpo $(X, \sqsubseteq)$ is of finite height h, the cpo $(\mathcal{P} \to \mathcal{P} \to X, \sqsubseteq)$ has height $|\mathcal{P}|^2 \cdot h$. Letting $\sqsubseteq$ denote also the point-wise extension of $\sqsubseteq$ to the cpo $\mathcal{P} \to \mathcal{P} \to X$, the least fixed-point can, in principle, be computed by finding the first identity in the chain $(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \Pi_\lambda(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \Pi_\lambda^2(\lambda p.\lambda q.\bot_\sqsubseteq) \sqsubseteq \cdots \sqsubseteq \Pi_\lambda^{|\mathcal{P}|^2 \cdot h}(\lambda p.\lambda q.\bot_\sqsubseteq)$. However, in the environment envisioned, such a computation is infeasible. The functions $(\pi_p : p \in \mathcal{P})$ defining Π_λ are distributed throughout the network, and, more importantly, even if the height h is finite, the number of principals $|\mathcal{P}|$, though finite, will be *very* large. Furthermore, even if resources were available to make this computation, we can not assume that any central authority is present to perform it.

This paper argues, by complementing the denotational model with sound *operational* techniques, that the situation is not as hopeless as we have suggested.

[2]Assuming that $(X, \preceq)$ is a lattice, which is often the case.

[3]Since the category of cpos and continuous functions has arbitrary products.

Our work essentially deals with the operational problems left as "future work" by Carbone *et al.* [6]. More specifically, this consists of three operational issues. Firstly, actual *computation* of trust values over a global, highly dynamic, decentralised network. Secondly, as pointed out previously [6], often it is infeasible and even unnecessary to compute the *exact* denotation of a set of policies, instead, a sound approximation of this value may be sufficient to make a trust-based decision. Finally, the inherently dynamic nature of the envisioned systems require algorithms and operational techniques that explicitly deal with the dynamic updating of trust policies (rather than simply dealing with updates by doing a *complete* re-computation of the trust-state).

Technically, we start by showing that although it may be infeasible to compute the function $\mathsf{gts} : \mathcal{P} \to \mathcal{P} \to X$, one can instead try to compute so-called *local* fixed-point values. We take the practical point-of-view of a specific principal R, wanting to reason about its trust value for another principal q. The basic idea is that instead of computing the entire function gts, and *then* "looking up" value $\mathsf{gts}(R)(q)$ to learn R's trust in q, one may instead compute this value directly. We derive from the setting of the model, a dependency graph which represents just the policy dependencies that are relevant for the particular computation. From there, we prove a convergence result that enables applicability of a robust totally-asynchronous distributed algorithm of Bertsekas [1] for fixed-point computation. This is developed in Section 2. In Section 3 we take very mild assumptions on the relation between the two orderings in trust structures. By using standard, order-based proof-techniques in the new setting of two distinct orderings, these assumptions allow us to derive two propositions. We present two efficient, distributed algorithms, which, due to the propositions, allow safe approximation of the fixed point. In some situations, this allows principals to take security-decisions without having to compute the exact fixed-point. For example, suppose we know a function, $m : \mathcal{P} \to \mathcal{P} \to X$, with the property that $m \preceq \mathsf{gts}$. In this case, if m is sufficient to allow a given request, so is the actual fixed-point. In Section 4, we address the problem dynamic policy-changes. We give an algorithm which seeks to maximise the reuse of information from "old" computations. For specific, commonly occurring, types of updates this is very efficient. For general updates, we give an algorithm which is better than the naive algorithm in many cases.

Related Work: Stephen Weeks has developed a mathematical framework suitable for modelling many existing trust-management systems [17]. The framework, which is a predecessor of the trust-structure framework, is based on defining a global trust-state ("authorisation map" [17]) by existence of least fixed-points of monotonic endo-functions on complete lattices.

The notion of trust structures [6,14] was developed to overcome the problems of partial information in trust-based security decision-making, inherent in large distributed systems, e.g. global-computing scenarios. It was argued that traditional trust-management approaches (e.g. [2,3,7,9,11] and a survey by Grandison *et al.* [10]) are sometimes too restrictive in environments of inherent partial information. The problem was addressed by introducing the notion of information into a framework similar to that of Weeks. The primary difference between the two frameworks is that in trust structures, least fixed-points are with respect to information, whereas in Weeks's framework they are with respect to trust (indeed, there is no notion of information ordering, and "trust" is identified with authorisation [17]). Another important difference is that in Weeks's framework, the trust policies (licenses) are carried by clients, instead of being stored at the "issuing" servers. This means that the operational approach is to let clients present along with their request, a set of licenses, which together give rise to what corresponds to function Π_λ. It is now the job of the server to (locally) compute the fixed-point, $\mathsf{lfp}\ \Pi_\lambda$, and decide how to respond. In contrast, in the trust-structure framework, the trust policies are naturally distributed. Each principal p, autonomously controls and stores its policy, π_p. This leads naturally to a distributed approach to computation of fixed-points, and this is indeed what we pursue in this paper.

The trust-structure framework has been further developed [13], providing a categorical axiomatisation of trust structures, and providing an understanding of the interval construction [6, 14] as a functor, which is the full and faithful left-adjoint in a co-reflection of a new category of trust structures, in a category of complete lattices. The trust-structure framework has a concrete instance in the SECURE project [4, 5] which deploys a specific class of trust structures, allowing probabilistic information, in its modelling of trust [13, 15].

The idea of computing *local* fixed-points has been recognised also by Vergauwen *et al.* in a non-distributed context of static program-analysis [16]. Dimitri Bertsekas has developed a substantial body of work on distributed- and parallel algorithms for fixed points, and this paper applies his asynchronous convergence theorem [1] to prove correctness of a distributed fixed-point algorithm. Finally, the EigenTrust system also defines its global trust-state by existence of unique (non order-theoretic) fixed-points [12], and the basic EigenTrust algorithm is essentially Bertsekas's globally synchronous algorithm.

2 Distributed Computation of Least Fixed-Points

In the framework of trust structures, a collection $\Pi = (\pi_p \mid p \in \mathcal{P})$ of continuous trust-policies defines a unique global trust-state, $\mathsf{gts} = \mathsf{lfp}\ \Pi_\lambda : \mathcal{P} \to \mathcal{P} \to X$, defining R's trust in q as $\mathsf{gts}(R)(q)$. We show in the following, how to compute the *local* fixed-point value, $\mathsf{gts}(R)(q)$, without having to compute the entire function gts. The reason for computing local values is that although the semantics of policies $\pi : (\mathcal{P} \to \mathcal{P} \to X) \to \mathcal{P} \to X$ allows policy π_R to depend on the values for *all* principals, we conjecture that, in practice, policies will not be written in this way. Instead, policies are likely to refer to a few known, and, to some extent, "trusted" principals. For the particular principal R wanting to compute its trust in another principal $q \in \mathcal{P}$, the set of principals that R's policy, π_R, *actually* depends on in its entry for q, is often a significantly smaller subset of $\mathcal{P}$. The idea is to compute, distributedly and dynamically, a dependency-graph which contains *only* the dependencies relevant for the computation of $\mathsf{gts}(R)(q)$, thus excluding a large set of principals that do not need to be involved in computation. Once this graph has been set up, we proceed with computation of $\mathsf{gts}(R)(q)$ by showing that the conditions of a general algorithmic convergence-theorem of Bertsekas [1] are satisfied.

We present our problem in the abstract setting of a distributed computation of the least fixed-point of a continuous endo-function on a cpo, and show then how our practical scenario maps into the abstract setting. We are given a cpo $(X, \sqsubseteq)$ of finite height h, and a natural number $n \in \mathbb{N}$. Writing $[n]$ for the set $\{1, 2, \ldots, n\}$, we have also a collection of n continuous functions $C = (f_i : i \in [n])$ of type $f_i : X^{[n]} \to X$. These functions induce a unique global continuous function $F = \langle f_i : i \in [n] \rangle : X^{[n]} \to X^{[n]}$ with the property that $\mathsf{Proj}_i \circ F = f_i$ for all $i \in [n]$. The function F then has a unique least-fixed-point, $\mathsf{lfp}\ F \in X^{[n]}$. We shall overload the symbol $\sqsubseteq$ to denote the ordering on X as well as the point-wise lifted ordering on $X^{[n]}$, i.e. $X^{[n]} \ni \bar{t} \sqsubseteq \bar{s} \in X^{[n]}$ iff for every $i \in [n]$ we have $\bar{t}(i) \sqsubseteq \bar{s}(i)$. Define a dependency graph $G_R = ([n], E)$, where $[n]$ is the set of nodes, and the edges, given as a function $E : [n] \to \mathbf{2}^{[n]}$, model (possibly an over-approximation of) the dependencies of the functions in C (i.e. we have $j \notin E(i)$ implies that function f_i does *not* depend on the value of "variable" j). We consider the nodes $[n]$ as network nodes that have memory and computational power, and where each node $i \in [n]$ is associated with function f_i. We assume that each node knows all nodes that it depends on, i.e. node i knows all edges $E(i)$. The node R is called the *root*. The goal of the distributed algorithms in this section is for the root node to compute its *local fixed-point value* $\mathsf{lfp}\ F_R$, that is, the value $(\mathsf{lfp}\ F)(R)$. Note that one trivial "distributed algorithm" for computing this is to send all the functions to the designated node R, and then let R compute the sequence $\perp^n, F(\perp^n), F^2(\perp^n), \ldots$. This may be acceptable in some scenarios, however there are issues one must consider. Firstly, nodes $[n]$ may not be willing to reveal their entire functions f_i. Secondly, we are not exploiting the potential parallelisation of the computation which may give a significant speed-up, and distribute the computational burdens. Thirdly, the encoding of a general function of type $X^n \to X$ requires $\Theta(|X|^n \log_2 |X|)$ bits.

We translate the trust-structure setting into our abstract setting by defining function f_R as policy π_R's entry for principal q. One then finds the dependencies of f_R by looking at which other policies this expression depends on[4]. If f_R depends

[4] If policies are written in a language such as suggested by Carbone *et al.* [6] there is a straightforward linear algorithm for computing the dependencies. In all cases, it is reasonable to assume that any $p \in \mathcal{P}$ knows the dependencies of π_p since p defines π_p.

on entry w in π_z then z is a node in the graph, and the function f_z is given by π_z's entry for w, with the dependencies of f_z given by the dependencies in the expression for w in π_z, and so on. From now on we shall work in the abstract setting as it simplifies notation. Note, that this definition might lead to a node z appearing several times in the dependency graph, e.g. with entries for principals w and y in π_z. We shall think of these as distinct nodes in the graph, although a concrete implementation would have node z play the role of two nodes, z_w and z_y. Note also, that the (minimal) dependency-graph is uniquely determined by the node functions f_i, and is *not* modelling any network topology. We will use phrases like x sends a message "via" edge (x, y) or that a message "passes through" edge (x, y). However, although the nodes of the graph represents concrete nodes in a physical communication-network, its edges do not represent any communication-links. We are thus assuming, in the spirit of global computing, an underlying physical communication-network allowing any node to send messages to any other node.

Throughout this paper, we use an asynchronous communication-model. The nodes communicate by asynchronous message-passing: we assume no known bound on the time it takes for a sent message to arrive. We assume that communication is reliable in the sense that any message sent eventually arrives, exactly once, unchanged, to the right node, and that messages arrive in the order in which they are sent. We assume that all nodes are willing to participate, and that they do not fail. The assumptions of non-failure and correctness of delivery ease the exposition, but we do not believe that they are essential, e.g. the asynchronous algorithm is often very robust [1].

Our algorithm for fixed-point computation consists of two stages. In the first stage, the dependency graph $G_R = ([n], E)$ is distributedly computed so that any node $i \in [n]$ knows i^+ and i^-. In the second stage, this information is used in a very simple asynchronous algorithm, which is described in Section 2.2.

2.1 Computing Dependencies

In this sub-section, we describe how the nodes distributedly compute the dependency graph described above. Two goals are to be fulfilled by the dependency computation. First, each node must obtain a list of the nodes that depend on it for the computation. Second, we want to compute a spanning tree $T_R \leq G_R$ with root R, so that each node knows its parent and its children in this tree. We denote $T_R = ([n], S)$, $S : [n] \to \mathbf{2}^{[n]}$, with $S(i) \subseteq E(i)$ for all $i \in [n]$. Note that we are not making use of this tree until Section 3.

For any node i, we denote the set $E(i)$ by i^+, and the set of nodes k for which $i \in E(k)$ (i.e. $E^{-1}(\{i\})$), by i^-. So to summarise, after the computation, any node i knows i^+ and i^-, and it knows its parent p_i and its children $S(i)$ in a spanning tree T_R rooted at R. Node i will store i^+ and i^- in variables of the same name, and will store p_i and $S(i)$ in variables $i.p$ and $i.S$.

The distributed algorithm for the dependency computation is described by a process that runs at each node. We use syntax inspired by process calculi to describe these processes. The semantics should be clear, perhaps except for the two constructs $||_I$ and **join-then**. Let L denote a set of labels (e.g. A,B,...). The construct $||_I$, for $I \subseteq L$, describes the parallel execution of $|I|$ processes (e.g. threads), where the behaviour of process $i \in I$ is described by an expression $i : Proc$, where $Proc$ is a process. The construct **join** J **then** $Proc$, with $J \subseteq L$, waits until each process $j \in J$ has terminated, and then executes process $Proc$. Note also that capital, italicised letters (e.g. X, Y, M, not $i.S$) are variables that become bound at reception of a message, e.g. **receive** (`mark`) **from** X is executed as soon as the reception of a `mark` message occurs, and in the following code, X is bound to the sender of that message.

Non-root nodes run a process given by Figure 1. The root node runs a special process which similar to that of Figure 1, but it has no parent, and it will initiate the computation. One way to think of the algorithm is as a simple distributed graph-marking algorithm: the initial message that a node i receives from a node j "marks" the node i, and j is then the "marker" for i. The edges between "marker" and "marked" nodes, will constitute the spanning tree T_R. Furthermore, once a node is marked it starts a "server" sub-process (labelled `A`) which accepts `mark`-messages from any node Y, adds Y to its dependency set i^-, and acknowledges with an "`ok`" message. A sub-process running in parallel (`B`), notifies all nodes that i depends on (i.e. i^+) of this dependency, and waits for each node to acknowledge. This acknowledgement is either "`ok`" in case i is not the marker, or "`marker`" in case i *is* the marker. Finally, when an acknowledgement has been received from each child, i acknowledges its "marker". Once the root node has received acknowledgement from each of its children, the algorithm terminates.

The following statements hold. The number of messages sent is $O(|E|)$, each message of bit length $O(1)$. This follows from the observation that for each edge in the graph there flows at most two messages, one `mark` and one acknowledgement. When the root node R has received acknowledgement from all its children then every node i, which is reachable from R, stores in the variable i^-, the set i^- (by abuse of notation), stores in variable $i.S$, the children of i in T_R, and in variable $i.p$, i's parent in T_R. Note, that we only mark the nodes that are reachable from R, which amounts to excluding any node that R does not depend on (directly or by transitivity) for computing its trust value for $q \in \mathcal{P}$.

2.2 Least-Fixed-Point Computation

For this section we assume that the dependency computation has already been run. We show that we are now in a situation in which we can apply existing work of Bertsekas for computation of the least fixed-point. Bertsekas has a class of algorithms, called totally asynchronous (TA) distributed iterative fixed-point algorithms, and a general theorem, which gives conditions ensuring that

Process: Dependency Algorithm for non-root node i

receive (`mark`) **from** X;

$i^- \leftarrow \{X\}$; $i.p \leftarrow X$; $i.S \leftarrow \emptyset$

$||_{\{\texttt{A,B}\}}$

`A` : **replicate**

[**receive** (`mark`) **from** Y;

$i^- \leftarrow i^- \cup \{Y\}$;

send (`ack`, `ok`) **to** Y]

`B` : $||_{c \in i^+}$

c : **send** (`mark`) **to** c;

receive (`ack`, M) **from** c;

if (M=`marker`) **then** $i.S \leftarrow i.S \cup \{c\}$

|| **join** i^+ **then send** (`ack`, `marker`) **to** X

Figure 1: Dependency Algorithm - Generic node behaviour

a specific TA algorithm will converge to the desired result. In our case, "converge to" will mean that each principal $i \in \mathcal{P}$ will compute a sequence of values $\perp_{\sqsubseteq} = i.t_0 \sqsubseteq i.t_1 \sqsubseteq \cdots \sqsubseteq i.t_k = (\mathsf{lfp}\ F)_i$. The general theorem is called the "Asynchronous Convergence Theorem" (ACT), and we use this name to refer to Proposition 6.2.1 of Bertsekas's book [1]. The ACT applies in any scenario in which the so-called "Synchronous Convergence Condition" and the "Box Condition" are satisfied. Intuitively, the synchronous convergence condition states that if algorithm is executed synchronously, then one obtains the desired result. In our case this amounts to requiring that the "synchronous" sequence $\perp_{\sqsubseteq} \sqsubseteq F(\perp_{\sqsubseteq}) \sqsubseteq \cdots$ converges to the least fixed-point, which is true. Intuitively, the box condition requires that one can split the set of possible values appearing during synchronous computation into a product ("box") of sets of values that appear locally at each node in the asynchronous computation.

We show that, as a consequence of monotonicity, the conditions of the Asynchronous Convergence Theorem are satisfied in our setting, and so, we can deploy a TA distributed algorithm. The algorithm is very simple, and consists simply of a process running at each node. Each process i will compute, asynchronously, its function f_i with respect to the best known estimates of the values of its dependencies. If computation of function f_i at node i results in a new "current" value $i.t_{cur} \in X$ (which always satisfies $i.t_{cur} \sqsubseteq (\mathsf{lfp}\ F)_i$), then node i broadcasts an update to all nodes that depend on this value. We show that the Asynchronous Convergence Theorem ensures that this process converges towards the right values at all nodes, and because of our assumption of finite height cpos, the distributed system will eventually reach a state which is stable. In this state, each node i will have computed $(\mathsf{lfp}\ F)_i$.

We will assume that after the dependency-graph algorithm has run, each node

i allocates variables $i.t_{cur}$ and $i.t_{old}$ of type X, which will later record the "current" value and the last computed value in X. Each node i has also an array, denoted by $i.m$. The array $i.m$ is of type X array, and will be indexed by the set i^+. The following concept of an *information approximation* is central in our results.

Definition 2.1. *Let $F : X^{[n]} \to X^{[n]}$ be continuous. Say that a value $\bar{t} \in X^{[n]}$, is an* information approximation for F *if $\bar{t} \sqsubseteq \mathsf{lfp}\ F$ and $\bar{t} \sqsubseteq F(\bar{t})$.*

Proposition 2.1. *Let $\bar{t}$ be any information approximation for F. Assume that after running the dependency-graph algorithm, the arrays of the nodes are initialised with $\bar{t}$, i.e. for all $i \in [n]$, and all $j \in i^+$ $i.m[j] = \bar{t}_j$, and $i.t_{old} = \bar{t}_i$. Then the Synchronous Convergence Condition and the Box Condition of the Asynchronous Convergence Theorem are both satisfied.*

Proof. Define a sequence of sets $X^{[n]} \supseteq \cdots \supseteq X(k) \supseteq X(k+1) \supseteq \cdots$ by

$$X(k) = \{m \in X^{[n]} \mid F^k(\bar{t}) \sqsubseteq m \sqsubseteq \mathsf{lfp}\ F\}$$

Note that $X(k+1) \subseteq X(k)$ follows from the fact that $F^k(\bar{t}) \sqsubseteq F^{k+1}(\bar{t})$ for any $k \in \mathbb{N}$, which, in turn, holds since $\bar{t}$ is an information approximation. For the synchronous convergence condition, assume that $m \in X(k)$ for some $k \in \mathbb{N}$. Since $F^k(\bar{t}) \sqsubseteq m \sqsubseteq \mathsf{lfp}\ F$, we get by monotonicity $F^{k+1}(\bar{t}) \sqsubseteq F(m) \sqsubseteq F(\mathsf{lfp}\ F) = \mathsf{lfp}\ F$. Let $(y_k)_{k\in\omega}$ be so that $y_k \in X_k$ for every k. Since X is of finite height there exists $k_h \in \mathbb{N}$ so that for all $k' \geq k_h$, $X(k') = \{\mathsf{lfp}\ F\}$. Thus any such $(y_k)_{k\in\omega}$ converges to $\mathsf{lfp}\ F$. The box condition is also easy:

$$X(k) = \prod_{i=1}^{n} \{m(i) \in X \mid m \in X^{[n]}, F^k(\bar{t}) \sqsubseteq m \sqsubseteq \mathsf{lfp}\ F\ \}$$

$\square$

In this section, we shall invoke the proposition in the trivial case of the information approximation $\bar{t} = \perp_{\sqsubseteq}^{n}$. Later, when considering dynamic algorithms for policy updates, we invoke the proposition with more interesting information approximations.

We briefly describe the asynchronous algorithm of Bertsekas. Each node i has the lists i^+ and i^-, resulting from dependency computation, and has also the variables $i.m$, $i.t_{cur}$ and $i.t_{old}$, mentioned earlier. Initially, $i.t_{cur} = i.t_{old} = \perp_{\sqsubseteq}$, and the array is also initialised with $\perp_{\sqsubseteq}$. For any i and $j \in i^+$, when i receives a message, which is always a value $t \in X$, from a node $j \in i^+$, it stores this message in $i.m[j]$. Any node is always in one of two states, *sleep* or *wake*. If a node is in the *sleep* state, the reception of a message triggers a transition to the *wake* state. All nodes start by making a transition from the *sleep* state to the *wake* state. In the *wake* state any node i repeats the following: it starts by computing its function with respect to the values in $i.m$. If there was no change in the resulting value (compared to the last value computed), it will go to the *sleep* state unless a message was received since it was last sleeping. Otherwise, if a new value resulted from the computation, this value is sent to all nodes in i^-. Concurrently with this, we run a termination detection algorithm, which will detect when all nodes are in the *sleep*-state, and no messages are in transit. Bertsekas has already addressed this problem [1], and his termination-detection algorithm directly applies.

This simple asynchronous behaviour gives a highly parallel, robust algorithm for which correctness follows from Proposition 2.1 and the Asynchronous Convergence Theorem. Furthermore, we can prove that since any node i sends values only when a change occurs, then by monotonicity of f_i, i will send at most $h \cdot |i^-|$ messages[5], each of size $O(\log_2 |X|)$ bits. Node i will receive at most $h \cdot |i^+|$ messages, and in the worst case it will do as many computations of f_i. Globally, the number of messages sent is $O(h \cdot |E|)$ each of bit size $O(\log_2 |X|)$. The cost of the termination-detection scheme must be added to this.

Note that a global invariant in this algorithm is that any value computed locally at a node, i.e. by $i.t_{cur} \leftarrow f_i(i.m)$, is a component in an information approximation for F. That is, it holds everywhere, at any time, that (1) $i.t_{cur} \sqsubseteq (\mathsf{lfp}\ F)_i$ and (2) $i.t_{cur} \sqsubseteq f_i(i.m)$. To see this, note that $(1, 2)$ hold initially, and that both properties are preserved by the update $i.t_{cur} \leftarrow f_i(i.m)$ whenever $i.m[y] \sqsubseteq (\mathsf{lfp}\ F)_y$ for all $y \in i^+$, which is always true. We state this fact as a lemma, as it becomes very useful in later sections, where we consider fixed-point approximation algorithms, and dynamics.

Lemma 2.1. *Any value $t \in X$ computed by any node $i \in [n]$ at any time in the algorithm by the statement $i.t_{cur} \leftarrow f_i(i.m)$, is a part of an information approximation, in the sense that $i.t_{cur} \sqsubseteq (\mathsf{lfp}\ F)_i$ and $t_{old} \sqsubseteq t_{cur}$.*

Finally, one might argue against doing trust computations in a distributed manner. There are implicit trust implications induced by the algorithm, i.e. i "trusts" $j \in i^+$, not only in the sense of relying on its policy, but also to perform correctly in the algorithm. We believe that this is an inherent property of the trust-structure model. We have argued that computing the fixed-point non-distributedly is inefficient (high communication to send policies, and no parallelism), and in violation with the privacy of principals. One solution to this could be something in between, in which some of the policies are sent to some designated nodes which will serve as "trusted computation servers", representing some subset of $[n]$, for the duration of the fixed-point computation.

[5]In fact, there will be *only* $O(h)$ *different* messages, each sent to all of i^-. Consequently, a broadcast mechanism could implement the message delivery efficiently.

3 Approximation techniques

In this section we develop two techniques for safe *approximation* of the fixed-point. Consider a situation in which a client principal p wants to access a resource controlled by server v. Assume that the access-control policy of v is that to allow access, its trust in p should be trust-wise above some threshold $t_0 \in X$, i.e. the fixed-point should satisfy $t_0 \preceq (\mathsf{lfp}\ \Pi_\lambda)(v)(p)$. The goal of the two approximation techniques is to allow the server to make its security decision *without* having to actually compute the entire fixed-point and then make the $\preceq$-check. Instead, the server is able to efficiently compute an element $\bar{t} : \mathcal{P} \to \mathcal{P} \to X$ which is related to the fixed point in such a way that the desired property can be asserted.

We need some preliminary terminology. Let $T = (X, \preceq, \sqsubseteq)$ be a trust structure, i.e. $(X, \sqsubseteq)$ is a cpo with bottom $\bot_\sqsubseteq$ and $(X, \preceq)$ is a partial order (not necessarily complete). We assume also that $(X, \preceq)$ has a least element, $\bot_\preceq$. Denote the $\sqsubseteq$-lubs/glbs by $\sqcup$ and $\sqcap$, and the $\preceq$-lubs/glbs by $\vee/\wedge$.

If for any countable $\sqsubseteq$-chain $C = \{x_i \in X \mid i \in \mathbb{N}\}$ and any $x \in X$ we have (i) $x \preceq C$ implies $x \preceq \bigsqcup C$ and (ii) $C \preceq x$ implies $\bigsqcup C \preceq x$, then $\preceq$ can be said to be $\sqsubseteq$-continuous.

3.1 Bounding "Bad Behaviour"

The first technique lets a client convince a server that its trust in the client is trust-wise above some level. The technique is based on the following proposition.

Proposition 3.1. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $t \in X^{[n]}$ and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. If $t \preceq \lambda k.\bot_\sqsubseteq$ and $t \preceq F(t)$ then $t \preceq \mathtt{lfp}_\sqsubseteq F$.*

Proof. We have $t \preceq \lambda k.\bot_\sqsubseteq$ which implies $F(t) \preceq F(\lambda k.\bot_\sqsubseteq)$ by $\preceq$-monotonicity. Since $t \preceq F(t)$, transitivity implies that $t \preceq F(t) \preceq F(\lambda k.\bot_\sqsubseteq)$. So again by $\preceq$-monotonicity of F and transitivity

$$t \preceq F(t) \preceq F^2(t) \preceq F^2(\lambda k.\bot_\sqsubseteq)$$

Now since for all $i \geq 0$ we have $t \preceq F^i(\lambda k.\bot_\sqsubseteq)$, the fact that $\preceq$ is $\sqsubseteq$-continuous implies that

$$t \preceq \bigsqcup_i F^i(\lambda k.\bot_\sqsubseteq) = \mathtt{lfp}_\sqsubseteq F$$

$\square$

This proposition is the basis of an efficient protocol for a kind of "proof-carrying authorisation". Consider for simplicity the "MN" trust-structure T_{MN} [13], which satisfies the information-continuity requirement. Recall that, in this structure, trust values are pairs (m, n) of natural numbers, with the orderings given by $(m,n) \sqsubseteq (m',n') \iff m \le m'$ and $n \le n'$, and $(m,n) \preceq (m',n') \iff m \le m'$ and $n \ge n'$.

Suppose principal p wants to efficiently convince principal v that v's trust value for p is a pair, (m,n), with the property that n is less than some $N \in \mathbb{N}$, thus giving v an upper bound on the "bad behaviour" of p. Let us assume that v's trust policy π_v depends on a large set S of principals. Assume also that it is sufficient that principals a and b in S have a reasonably "good" value for p, to ensure that v's value for p is not too "bad". An example policy with this property could be written in the language of Carbone *et al.* [6] as

$$\pi_v \equiv \lambda x : \mathcal{P}.(\ulcorner a \urcorner(x) \wedge \ulcorner b \urcorner(x)) \vee \bigwedge_{s \in S} \ulcorner s \urcorner(x)$$

This policy, informally, says that any x should have "high" trust with a and b, or, with all of $s \in S$, for the v to assign a "high" value to x.

If p knows that it has previously performed well with a and b, and knows also that v depends in this way on a and b, it can engage in the following protocol. Principal p sends to v the trust values $t = [p \mapsto (0,N), a \mapsto (0,N_a), b \mapsto (0,N_b)]$. Upon reception, v first extends t to a global trust state $\bar{t}$, which is the extension of t to a function of type $\mathcal{P} \to \mathcal{P} \to T_{MN}$, given by

$$\bar{t} = \lambda x \in \mathcal{P} \lambda y \in \mathcal{P}. \begin{cases} (0,N) & \text{if } x = v \text{ and } y = p \\ (0,N_a) & \text{if } x = a \text{ and } y = p \\ (0,N_b) & \text{if } x = b \text{ and } y = p \\ (0,\infty) & \text{otherwise} \end{cases}$$

Principal p wants to verify that $\bar{t}(x)(y) \preceq \bot_{\sqsubseteq} = (0,0)$ for all x,y. But this holds trivially if $y \ne p$ or $x \ne v,a,b$ because then $\bar{t}(x)(y) = (0,\infty) = \bot_{\preceq}$. For the other few entries it is simply an order-theoretic comparison $\bar{t}(x)(y) \preceq (0,0)$. Now v tries to verify that $\bar{t} \preceq \Pi_\lambda(\bar{t})$. To do this, v verifies that $(0,N) \preceq \pi_v(\bar{t})(p)$. If this holds then it sends the value t to a and b, and ask a and b to perform a similar verification, e.g $(0,N_a) \preceq \pi_a(\bar{t})(p)$. Then a and b reply with *yes* if this holds and *no* otherwise. If both a and b reply *yes*, then p is sure that $\bar{t} \preceq \Pi(\bar{t})$: by the checks made by v, a and b, we have that $\bar{t}(x)(y) \preceq \Pi_\lambda(\bar{t})(x)(y)$ holds for pairs $(x,y) = (v,p),(a,p),(b,p)$, but for all other values it holds trivially since $\bar{t}$ is the $\preceq$-bottom on these. So by Proposition 3.1, we have $\bar{t} \preceq \mathsf{lfp}\,\Pi_\lambda$, and so v is *ensured* that its trust value for p is $\preceq$-greater than $(0,N)$.

We have illustrated the main idea of the protocol by way of an example. In general, the proof t, may include a large number of principals, which would then have to be involved in the verification process.

The approximation protocol has very much the flavour of a proof-carrying authorisation: the requester (or prover) must provide a proof that its request should be granted. It is then the job of the service-provider (or verifier) to check that the proof is correct. The strength of this protocol lies in replacing an entire fixed-point computation with a few local checks made by the verifier, together with a few checks made by a subset of the principals that the verifier depends on. A nice property of this protocol is that part of the information that the prover needs to supply should be available locally to the prover – it should already know who it has performed well with in the past. There are, however, two important restriction of this approach. First, as in the example, in order to construct its proof, the prover needs information about the verifiers trust policy, and of the policies of those whom the verifier depends on. If policies are secret, it is not clear how the verifier would construct this proof. Secondly, because of the requirement that $t \preceq \bot_{\sqsubseteq}$, the approach can usually only be used to prove properties stating "not too much bad behaviour", and not properties guaranteeing sufficient good behaviour. Notice that the protocol for exploiting this proposition has a message complexity which is independent of the height of the lattice. In contrast, the algorithm for computing fixed-points has message complexity $O(h \cdot |E|)$. We present now another approach which requires more computation and communication, but does not have the two mentioned restrictions.

3.2 Exploiting Information Approximations

The approximation technique developed in this section is different from that of the "proof-carrying authorisation"-protocol of the previous section. The protocol of this section does not require the "prover" to provide any information. Instead, it can be seen as a merge between the fixed-point computation-algorithm from Section 2.2, and the proof-checking technique from the previous section. The technique of this section, allows for "verifiers" to compute an information-approximation to the fixed point by finding a "snapshot" of the "current" values in the fixed-point algorithm. The nodes then make a collection of local checks on this snapshot, in order to infer that the fixed-point value must be trust-wise above the current value.

Let $F : X^{[n]} \to X^{[n]}$ be $\sqsubseteq$-continuous. Recall that a value $t \in X^{[n]}$, is an *information approximation* for F, if $t \sqsubseteq \mathsf{lfp}\ F$ and $t \sqsubseteq F(t)$. The following proposition is the basis for our approximation protocol.

Proposition 3.2. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $t \in X^{[n]}$ and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. Assume that t is an information approximation for F, and that $t \preceq F(t)$, then $t \preceq \mathsf{lfp}\ F$.*

Proof. Since t is an information approximation for F, we have by easy induction that for all $k \in \mathbb{N}$, $F^k(t) \sqsubseteq F^{k+1}(t) \sqsubseteq \mathsf{lfp}\ F$, and so by continuity of F, $\bigsqcup_{k\in\mathbb{N}} F^k(t) = \mathsf{lfp}\ F$. Since $t \preceq F(t)$, an easy induction gives $t \preceq F^k(t)$ for all k. Then the information continuity of $\preceq$ implies that $t \preceq \mathsf{lfp}\ F$. $\square$

This proposition gives the basis for exploiting values that are information approximations for F. This is very useful because, by Lemma 2.1, a global invariant in the asynchronous fixed-point algorithm is that all values computed, and thus also values transmitted on communication channels, are information approximations for F. This means that we can combine the algorithm with a protocol that, intuitively, implements the check for the condition $t \preceq F(t)$ in the above proposition.

Imagine that during the execution of the asynchronous algorithm, there is a point in time in which no messages are in transit, all nodes have computed their function, and sent the value to all that depend on it. Thus we have a "consistent" state in the sense that for any node x and any node $y \in x^+$ then $x.m[y] = y.t_{cur}$. In particular if x, z both depend on y, then they agree on y's value: $x.m[y] = z.m[y] = y.t_{cur}$. In this ideal state, there is a consistent vector t, which is an information approximation for F, containing the value t_x for nodes $x \in [n]$, i.e. $t_x = x.t_{cur}$. If the state of the distributed system was frozen at this point, and all nodes x, simultaneously make the check $x.t_{cur} \preceq f_x(x.m)$, then vector t satisfies $t \preceq F(t)$. By Proposition 3.2, the root node R then knows $t_R \preceq \mathsf{lfp}\ F_R$, which is what we want. Of course, this ideal situation would rarely occur in a real execution, except for when the algorithm terminates, in which case, the conclusion is trivial since $t = \mathsf{lfp}\ F$. The aim of the algorithm is to enforce a consistent view of such an ideal situation during execution of the asynchronous algorithm, i.e. fix a vector t, ensure that this vector is consistent, and then make all the checks $x.t_{cur} \preceq f_x(x.m)$ for all $x \in [n]$. This is a kind of so-called snapshot-algorithm (see Bertsekas [1]), in which the (local views of the) global state of the system is recorded during execution of an algorithm. It is slightly less complicated since we are not interested in the status of communication links, but slightly more complicated since each snapshot-value must be propagated to a specific set of nodes.

We describe now a distributed algorithm implementing this. We assume that the asynchronous algorithm is running, and at some point the root node decides to run the approximation check (e.g. because it has computed a (non fixed-point) value $R.t_{cur}$ which is sufficient to allow access). We assume that each node $i \in \mathcal{P}$ has additional variables $i.t_{app} : X$ and $i.m_{app} : X$ array, indexed by i^+. The array will eventually store only consistent values. The algorithm, as usual, consists of a special process run by the root, and another similar process running at non-root nodes, given by Fig. 2.

Recall, that the dependency-graph algorithm has generated also a spanning tree T_R, rooted at R. The root initiates the approximation algorithm. It starts by sending an init message to each of it's children listed in $R.S$ (Fig. 2, label $\mathsf{A1}$). Now it waits until it is in a *locally consistent* state ($\mathsf{A2}$), which means that, in the asynchronous algorithm, it has just computed $R.t_{cur} \leftarrow f_R(R.m)$, and (if necessary) has sent that value to each of R^-. Once in such a state, R saves the value by doing $R.t_{app} \leftarrow R.t_{cur}$ – this value will become the value for R in the

Process: non-root nodes i

$||_{\{\mathtt{A},\mathtt{B},\mathtt{C}\}}$

    A : **receive** (init);

        $||_{\{\mathtt{A1},\mathtt{A2}\}}$

            A1: $||_{c \in i.S}$ c : **send** (init) **to** c;

            A2: [$//w$ait until consistent state];

                $i.t_{app} \leftarrow i.t_{cur}$;

                $||_{j \in i^-}$ j : **send** (copy) **to** j;

    B : $||_{\{\mathtt{B1},\mathtt{B2}\}}$

            B1: $||_{k \in i^+}$

                    k : **receive** (copy) **from** k;

                        $i.m_{app}[k] \leftarrow i.m[k]$;

            B2: **join** $\{\mathtt{B1}, \mathtt{A2}\}$ **then**

                $i.b$: bool $\leftarrow (i.t_{app} \preceq f_i(i.m_{app}))$;

    C : $||_{\{\mathtt{C1},\mathtt{C2}\}}$

            C1: $||_{c \in i.S}$

                    c : **receive** ($i.b_c$: bool) **from** c;

            C2: **join** $\{\mathtt{C1}, \mathtt{B2}\}$ **then**

                send $(i.b \wedge (\bigwedge_{c \in i.S} i.b_c))$ **to** $i.p$;

Figure 2: Snapshot Algorithm - Generic node behaviour

consistent vector we are seeking. R now sends a copy message to each node in R^- (A2). A node $y \in R^-$ which receives a copy message from R will copy the last value received from R into its approximation array, i.e. $y.m_{app}[R] \leftarrow y.m[R]$ (B1). Since we are assuming a reliable network, the copied value is $R.t_{app}$, and so we are propagating consistent values. Root R now waits until each node $z \in R^+$ has sent a copy message, and computes then $R.t_{app} \preceq f_R(R.m_{app})$ (B2). Finally, the root waits for all children in the spanning tree to have replied with a boolean, and if all of these are true and the check succeeded (C1, C2), then the root is ensured that $R.t_{app} \preceq (\mathsf{lfp}\ F)_R$. Non-root nodes i, once initiated, do almost the same. The only difference is that after the check has been made, and all children in the spanning tree have replied with a boolean, i sends value true to its parent $i.p$ only if all $i.S$ sent true and i's own check succeeded.

Since there is a constant number of messages sent for each edge in G_R, the message complexity of the snapshot algorithm is $O(|E|)$ messages, each of size $O(1)$ bits.

A useful property of this algorithm is that it can be run concurrently with the asynchronous fixed-point algorithm - there is no reason to stop! One may simply allocate a thread implementing the approximation-check, which runs concurrently with the asynchronous fixed-point algorithm.

Note that, the style of this protocol is different than that of the previous section. In the previous protocol the client presents a "proof" t which the servers then verifies. It is not clear how one could use Proposition 3.2 in this style. In particular, if a client presented a "proof" t, then it is not clear how the servers would check that $t \sqsubseteq \mathsf{lfp}\ F$ without already knowing $\mathsf{lfp}\ F$.

3.3 Dual Propositions and Generalisation

Note that both the propositions in this section have "dual" versions.

Proposition 3.3. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $t \in X^{[n]}$ and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. If $\perp_\sqsubseteq \preceq t$ and $F(t) \preceq t$ then $\mathsf{lfp}\ F \preceq t$.*

Proposition 3.4. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $t \in X^{[n]}$ and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. Assume that t is an information approximation for F, and that $F(t) \preceq t$. Then $\mathsf{lfp}\ F \preceq t$.*

We can deploy similar algorithms for the duals. At first sight, Proposition 3.3 does not seem as useful as its dual. The conclusion $\mathsf{lfp}\ F \preceq t$ can usually only be used to *deny* a request, and a prover in the protocol for Proposition 3.3 would probably not be interested in supplying information which would help "refuting" its claim. However, this is not always so. For example, if one is using trust structures conveying probabilistic information (e.g. [4, 15]), an assertion of the form $\mathsf{lfp\ F} \preceq m$, can convince the verifier that when interacting with the prover the probability of a "bad" outcome is below some threshold.

We can use essentially the same algorithm as that of Section 3.2, for exploiting Proposition 3.4. Servers could incorporate the check $F(t) \preceq t$ together with the dual check $t \preceq F(t)$. Thus, the root could fairly refute a request without actually computing the fixed point. Note the particular case in which both checks are satisfied. Since $\preceq$ is a partial ordering of X, this will occur just in case $t = \mathsf{lfp\ F}$, and so could serve as an alternative termination-detection mechanism.

One can note that the two approximation techniques are merely instances of the same general proposition.

Proposition 3.5. *Let $(X, \preceq, \sqsubseteq)$ be a trust structure in which $\preceq$ is $\sqsubseteq$-continuous. Let $\bar{p} \in X^{[n]}$ and $F : X^{[n]} \to X^{[n]}$ be any function that is $\sqsubseteq$-continuous and $\preceq$-monotonic. Assume that $\bar{p}$ satisfies $\bar{p} \preceq F(\bar{p})$. If there exists an information approximation $\bar{t} \in X^{[n]}$ for F, with property that $\bar{p} \preceq \bar{t}$, then $\bar{p} \preceq \mathsf{lfp}\ F$.*

Proof. The proof of Proposition 3.5 is similar to that of Proposition 3.1. We use the diagram:

$$\begin{array}{ccccccccc} \bar{p} & \preceq & F(\bar{p}) & \preceq & \cdots & \preceq & F^i(\bar{p}) & \preceq & \cdots \\ \curlyeqprec \!\!\!\!\! \mid & & \curlyeqprec \!\!\!\!\! \mid & & & & \curlyeqprec \!\!\!\!\! \mid & & \cdots \\ \bar{t} & \sqsubseteq & F(\bar{t}) & \sqsubseteq & \cdots & \sqsubseteq & F^i(\bar{t}) & \sqsubseteq & \cdots \end{array}$$

By continuity of $\preceq$ we have $\bar{p} \preceq \bigsqcup_i F^i(\bar{t})$. □

Note that one obtains Proposition 3.1 with the trivial information approximation $\bar{t} = \bot_{\sqsubseteq}$, and Proposition 3.2 by taking the proof to be the approximation, i.e. $\bar{p} = \bar{t}$.

We note finally, that the $\sqsubseteq$-continuity property, required of $\preceq$ in Proposition 3.5, is satisfied for all interesting trust-structures we are aware of: Theorem 3 of Carbone *et al.* [6] implies that the information-continuity condition is satisfied for all interval-constructed structures. Furthermore, their Theorem 1 ensures that interval-constructed structures are complete lattices with respect to $\preceq$ (thus ensuring existence of $\bot_{\preceq}$). Several natural examples of non-interval domains can also be seen to have the required properties [13]. The requirement that all policies π_p are monotonic also with respect to $\preceq$ is reasonable. Intuitively, it amounts to saying that if everyone raises their trust-levels in everyone, then policies should not assign lower trust levels to anyone.

4 Dynamics

In this section, we consider what one might do in case of some function f_i, dynamically changing to f'_i, denoted $f_i \mapsto f'_i$.

Suppose that the fixed-point computation has terminated, i.e. each node x knows its value t_x along with values t_y for $y \in x^+$, so that for all x, $t_x = (\mathsf{lfp}\ F)(x)$. Suppose now that some i makes a policy update, i.e. changes its function f_i to f_i', e.g. due to new information being available. One could now let f_i broadcast a "reset"-message to all nodes and computation (including dependency graph discovery) could restart. However, in this approach there is a lot of information which is unnecessarily discarded. For some special (and commonly occurring) types of updates one can be more efficient in the reuse of information. A very simple example of this is when the update is information increasing and doesn't change the structure of the dependency graph. By information increasing, we mean that $f_i \sqsubseteq f_i'$. In this case, denoting $F' = F[f_i'/i] = \langle f_1, f_2, \ldots, f_{i-1}, f_i', f_{i+1}, \ldots f_n \rangle$, we have $F \sqsubseteq F'$, and so $\mathsf{lfp}\ F \sqsubseteq \mathsf{lfp}\ F'$. It is easy to see that the values $(t_x)_{x \in [n]}$ are an information approximation also to F'. This means that one can invoke Prop. 2.1, and so the algorithm can continue with the old values.

In many systems, it is likely that observing interactions between principals will cause information-increasing changes in policies [13,15]. However, also more general types of updates will occur, but we conjecture that in many systems they will be less frequent, i.e. policy changes are rare whereas obtaining new information about behaviour is not, but both trigger a change in the trust-policy function. None the less, occasionally these non-increasing changes will occur, and so they must be handled as efficiently as possible.

4.1 Non Edge-deleting Updates

Consider an update, $f_i \mapsto f_i'$, where the dependencies of f_i are contained in the dependencies of f_i', i.e. node i adds additional edges to the dependency graph, but deletes none. Clearly, one must extend the current dependency graph to a larger graph in order to proceed with computation. Furthermore, since function f_i has changed, any node j that depends on i, either directly or indirectly, will have inconsistent values in their arrays $j.m$. The idea in our algorithm is for those nodes (and only those nodes!) to take on a safe approximation to the updated function. Since the update can be arbitrary, we choose value $\perp_{\sqsubseteq}$ as our approximation, to ensure that this value is, in fact, an information approximation. Once all these nodes have taken a safe approximation, we can invoke our Proposition 2.1, to ensure that the Asynchronous Convergence Theorem is satisfied, and computation can proceed in the updated graph.

Concretely, node i will now run two algorithms concurrently. Firstly, i will run, in the role of a root node, a generalised version of the dependency-graph algorithm from Section 2. More specifically, it is the same algorithm, except that we allow generalised acknowledgement-messages which carry values from X. The reason is that i might initiate a new node j, which has an edge to a node k, which was already in the old dependency graph (see Figure 3). In this case, there is no

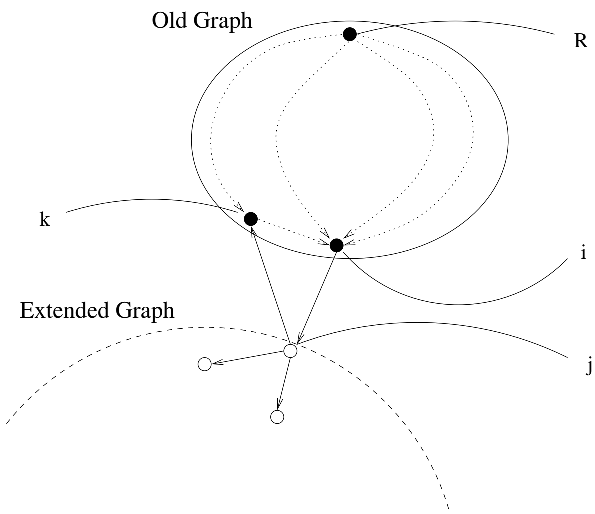

Figure 3: Situation where a new node i has an edge back to an old node k. Old nodes are black and new nodes white. Edges in the new graph are full lines, whereas dotted lines denote paths in the old graph.

reason for j to assume $\bot_{\sqsubseteq}$ for k, instead, k acknowledges and sends its current value, $k.t_{cur}$. This leads to a sound approach when combined with the second algorithm.

In the second algorithm, the node i ensures that all nodes that depend on it, either directly or indirectly, take on a safe approximation to the new function. This algorithm is given by Figure 4. Node i will set $i.t_{cur} \leftarrow i.t_{old} \leftarrow \bot_{\sqsubseteq}$, while leaving its array $i.m$ unchanged. The algorithm will terminates by a principle similar to that of the dependency-graph algorithm. In contrast to that algorithm, the messages "flow" in the opposite direction of the edges, e.g. from i to i^-. Node i starts by sending `reset` messages to each node $j \in i^-$. It expects to receive a reply from each of these nodes, and once received, the algorithm is terminated. Any node j (including possibly i itself) which receives a `reset` message from a node k, starts by setting its entry for k, $j.m[k] \leftarrow \bot_{\sqsubseteq}$. It then sets $j.t_{old} \leftarrow j.t_{cur} \leftarrow \bot_{\sqsubseteq}$, which is a safe approximation of its value, with respect to the updated fixed point. Unless $i = j$, node j then propagates the `reset` message to each $j' \in j^-$. The propagation is only done for the first `reset` message received. Subsequent message are simply acknowledged immediately after the array entry has been updated. Once j has received acknowledgements from each of its "parents" (i.e. j^-), it either sends an acknowledgement to the first node which sent it the `reset` message, or, in case of $i = j$, it stops. The effect of this algorithm is the following. For each node $j \in [n]$, j will receive a `reset` message from each $l \in j+$ which has a path to i. Each of the array entries for these nodes, $j.m[l]$, is set to $\bot_{\sqsubseteq}$ in order to take a safe approximation to the updated fixed point. Note however, that any

Process: nodes $j(\neq i)$

receive (`reset`) **from** X;

$j.t_{old} \leftarrow j.t_{cur} \leftarrow j.m[X] \leftarrow \perp_{\sqsubseteq}$;

$||_{\{\mathtt{A},\mathtt{B}\}}$

`A` : **replicate**

[**receive** (`reset`) **from** Y;

$j.m[Y] \leftarrow \perp_{\sqsubseteq}$;

send (`ack`) **to** Y]

`B` : $||_{k \in j^-}$

k : **send** (`reset`) **to** k;

receive (`ack`) **from** k;

$||$ **join** j^- **then send** (`ack`) **to** X

Figure 4: Reset Algorithm - Generic node behaviour

node n which does not have a path to i, will never receive a `reset` message, and thus any $n' \in n^-$ which depends on n, will not reset its entry for n.

Finally, one must consider how the two algorithms work concurrently. Say that a node is "new" if it is in the dependency graph for the updated function F' but not in that for F. Similarly an "old" node is one that is in the dependency graph for F. Suppose first that i initiates a new node j, which depends on an old node k. If j informs k of the dependency before k "is reset" (receives a `reset` message), then j will receive the current value of k, which can later be reset to $\perp_{\sqsubseteq}$ if the `reset`-algorithm requires it (see Figure 3). Suppose instead that k has received its `reset` message before it is informed of j's dependency. In this case j will receive $\perp_{\sqsubseteq}$. In either case, when *both* algorithms have terminated, then we have extended the dependencies to the new dependency graph, and furthermore, any node j in the extended graph stores in its current value, and in its array $j.m$ only information approximations to the fixed point value. In this case, by Prop. 2.1 the conditions of the Asynchronous Convergence Theorem are satisfied, and computation with respect to the new function $F' = F[f_i'/f_i]$ can proceed.

In the worst case, in which every other node depends on i, this algorithm will reduce to the trivial "reset" algorithm, in which the `reset` message is broadcast to all nodes. Is not hard to see that the global number of messages sent in this algorithm is $O(|E'|)$, where E' denotes the edges in the extended graph. Each message has bit-size $O(1)$, or $O(\log_2 |X|)$ in case of generalised acknowledgements.

4.2 Non Edge-adding Updates

Consider an update, $f_i \mapsto f_i'$, where the dependencies of f_i are a super set of the dependencies of f_i', i.e. node i deletes edges, but adds none. In some ways, this case is simpler. One could just let i send a `delete` message to each j that "was deleted". Then i simply runs the "reset"-algorithm described in the previous section. This is safe, but one might argue that we might still have redundant "dangling" edges in the graph. This occurs in the case where a deleted edge ij disconnects from the root node, a set of nodes k, which are reachable only from j. If such disconnected k has a dependency back to a node l, which is still reachable from the root, then l will be sending values to k even though k is really not needed in order for the root to compute its value. It is not clear, short of a complete re-computation of the dependency graph (which could be acceptable), how to efficiently (dynamically) deal with this problem.

4.3 General Updates

It should be clear that a combination of these two algorithms can be used to recover from arbitrary updates, leaving still the problem of "dangling" edges. One simply initiates, a concurrent execution of the algorithms for updating the dependency graph (deletion and addition of edges), together with the reset-algorithm described previously.

5 Conclusion

We have presented concrete algorithmic techniques for computation and approximation of the least fixed-point of a collection of continuous functions on trust structures. We have shown that the assumptions of the Asynchronous Convergence Theorem of Bertsekas are satisfied if one initiates computation with a consistent information-approximation, which means that we can apply a well-established asynchronous fixed-point algorithm for both approximation and computation of the least fixed-point. We have considered trust structures in which the two orderings are related in that the information ordering is continuous with respect to the trust ordering. For these trust structures, we have proved two propositions which relate the two orderings, allowing one to reason about the least fixed-point of continuous functions that are also monotonic with respect to the trust ordering. The propositions are theoretically simple, but their novelty lies in that we are relating the two different orderings in a way that gives rise to efficient protocols that allows principals to reason about the fixed-point values without having to compute the exact fixed-point. The second approximation technique (Prop. 3.2) relies on an algorithm which supplies an information approximation. This is used to reason about the trust-relation between the current value ($i.t_{cur}$) and the actual fixed-point value $(\mathsf{lfp}\ F)_i$. This gives a nice connection between the asynchronous fixed-point algorithm, which automatically provides information approximations, and the idea of safe fixed-point approximation. In the final section, we have presented techniques to deal with dynamic updates of the policy functions, f_i. For information-increasing updates, this is very efficient as all current estimates are still valid (invoking Prop. 2.1), in the sense that that the asynchronous algorithm will converge to the correct value. For completely general updates, we have given an algorithm which does not discard the information that is definitely not affected by the update.

One can imagine the system starting from scratch, i.e. there is a collection of nodes, each with a trivial policy $\pi = \perp_{\sqsubseteq}$, implying that no nodes are connected in the dependency graph. As the system evolves, edges are added, reflecting creation of new trusting relationships. One runs the dynamic algorithms for policy updates to ensure that the computations are always consistent. The techniques are dynamic, which allows new nodes to enter and leave the network, without affecting the algorithms.

Apart from its application in implementing trust-structure-based systems, the technique for fixed-point computation presented in this paper is general enough to be used for order-theoretic fixed-point computation in any cpo with bottom, or complete lattice. In particular, the techniques could be the basis of a distributed implementation of a variant of Weeks's model of trust-management systems [17], in which credentials are stored by the issuing authorities instead of being presented by clients. This would allow also for revocation, implemented simply as a trust-policy update at the authority revoking the credential.

Interesting future work is to explore to which extent the area of abstract interpretation [8] can be applied for trust-structure fixed-points, e.g. using widening and narrowing to speed up computation, and allow possibly infinite height cpos. Also, it could be interesting to try and analyse the amortized complexity of our system. For example, if principal R wants to know its trust in q, it can run the algorithm presented in this paper to compute this value. Now, after some time has passed, principals might have made additional observations about q. Supposing that R at some point later wants to compute its trust in q, then since one reuses the information gained from the last computation, the second re-computation would be significantly faster. In general, one might consider the amortized cost of a sequence of operations which are either 'computation of the fixed-point value', or 'policy update'.

Acknowledgements. This work was done as part of the SECURE project, EU FET-GC IST-2001-32486, and we would like to thank everyone in the consortium for their cooperation. We thank also Vladimiro Sassone and Mogens Nielsen for useful feedback.

References

[1] D. P. Bertsekas and J. N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall International Editions. Prentice-Hall, Inc., 1989.

[2] M. Blaze, J. Feigenbaum, and A. D. Keromyti. KeyNote: Trust management for public-key infrastructures. In *Security Protocols: 6th International Workshop, Cambridge, UK, April 1998. Proceedings.*, volume 1550, pages 59–63, 1999.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings of the 17th Symposium on Security and Privacy, Oakland, California*, pages 164–173. IEEE Computer Society Press, 1996.

[4] V. Cahill and et al. Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, 2003.

[5] V. Cahill and J.-M. Signeur. Secure environments for collaboration among ubiquitous roaming entities. Website: `http://secure.dsg.cs.tcd.ie`, 2004.

[6] M. Carbone, M. Nielsen, and V. Sassone. A formal model for trust in dynamic networks. In *Proceedings from Software Engineering and Formal Methods, SEFM'03*. IEEE Computer Society Press, 2003.

[7] Y.-H. Chu, J. Feigenbaum, B. LaMacchia, P. Resnick, and M. Strauss. REFEREE: Trust management for (web) applications. *Computer Networks and ISDN Systems*, 29(8–13):953–964, 1997.

[8] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, Los Angeles, California, 1977. ACM-Press, New York.

[9] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomsa, and T. Ylonen. SPKI Certificate Theory. RFC 2693, ftp-site: `ftp://ftp.rfc-editor.org/in-notes/rfc2693.txt`, September 1999.

[10] T. Grandison and M. Sloman. A survey of trust in internet applications. *IEEE Communications Surveys & Tutorials*, 3(4), 2000.

[11] T. Jim. SD3: A trust management system with certified evaluation. In *Proceedings from the IEEE Symposium on Security and Privacy, Oakland, California*, pages 106–116, 2001.

[12] S. D. Kamvar, M. T. Schlosser, and H. Garcia-Molina. The eigentrust algorithm for reputation managments in P2P networks. In *Proceedings of the twelft international conference on World Wide Web, Budapest, Hungary*, pages 640–651, 2003.

[13] K. Krukow. On foundations for dynamic trust management. Unpublished PhD Progress Report, available at: `http://www.brics.dk/~krukow/`, 2004.

[14] M. Nielsen and K. Krukow. Towards a formal notion of trust. In *Proceedings of the 5th ACM SIGPLAN international conference on Principles and Practice of Declarative Programming*, pages 4–7. ACM Press, 2003.

[15] M. Nielsen and K. Krukow. On the formal modelling of trust in reputation-based systems. To be published in Springer Lecture Notes in Computer Science. Available online: `http://www.brics.dk/~krukow/`, 2004.

[16] B. Vergauwen, J. Wauman, and J. Lewi. Efficient fixpoint computation. In B. Le Charlier, editor, *Static Analysis (SAS'94)*, pages 314–328. Springer, Berlin, Heidelberg, 1994.

[17] S. Weeks. Understanding trust management systems. In *Proceedings from the IEEE Symposium on Security and Privacy, Oakland, California*, pages 94–106, 2001.

Recent BRICS Report Series Publications

RS-04-16 Karl Krukow and Andrew Twigg. ***Distributed Approximation of Fixed-Points in Trust Structures*****. September 2004. 25 pp.**

RS-04-15 Jesús Fernando Almansa. ***Full Abstraction of the UC Framework in the Probabilistic Polynomial-time Calculus ppc*****. August 2004.**

RS-04-14 Jesper Makholm Byskov. ***Maker-Maker and Maker-Breaker Games are PSPACE-Complete*****. August 2004. 5 pp.**

RS-04-13 Jens Groth and Gorm Salomonsen. ***Strong Privacy Protection in Electronic Voting*****. July 2004. 12 pp. Preliminary abstract presented at Tjoa and Wagner, editors,** ***13th International Workshop on Database and Expert Systems Applications*****, DEXA '02 Proceedings, 2002, page 436.**

RS-04-12 Olivier Danvy and Ulrik P. Schultz. ***Lambda-Lifting in Quadratic Time*****. June 2004. 34 pp. To appear in** ***Journal of Functional and Logic Programming*****. This report supersedes the earlier BRICS report RS-03-36 which was an extended version of a paper appearing in Hu and Rodríguez-Artalejo, editors,** ***Sixth International Symposium on Functional and Logic Programming*****, FLOPS '02 Proceedings, LNCS 2441, 2002, pages 134–151.**

RS-04-11 Vladimiro Sassone and Paweł Sobociński. ***Congruences for Contextual Graph-Rewriting*****. June 2004. 29 pp.**

RS-04-10 Daniele Varacca, Hagen Völzer, and Glynn Winskel. ***Probabilistic Event Structures and Domains*****. June 2004. 41 pp. Extended version of an article to appear in Gardner and Yoshida, editors,** ***Concurrency Theory: 15th International Conference*****, CONCUR '04 Proceedings, LNCS, 2004.**

RS-04-9 Ivan B. Damgård, Serge Fehr, and Louis Salvail. ***Zero-Knowledge Proofs and String Commitments Withstanding Quantum Attacks*****. May 2004. 22 pp.**

RS-04-8 Petr Jančar and Jiří Srba. ***Highly Undecidable Questions for Process Algebras*****. April 2004. 25 pp. To appear in Lévy, Mayr and Mitchell, editors,** ***3rd IFIP International Conference on Theoretical Computer Science*****, TCS '04 Proceedings, 2004.**