Accepted for 3rd International Conference on Policies for Distributed Systems (POLICY 2002), June 2002

Policies in Accountable Contracts

Brian Shand, Jean Bacon
University of Cambridge Computer Laboratory
JJ Thomson Avenue, Cambridge CB3 0FD, UK
email : {Firstname.Lastname}@cl.cam.ac.uk

Abstract

In this paper, accounting policies explicitly control resource usage within a contract architecture. Combined with a virtual resource economy, this allows efficient exchange of high-level computer services between untrustworthy participants. These services are specified as contracts, which must be signed by the participants to take effect. Each contract expresses its accounting policy using a limited language, with high expressiveness but predictable execution times. This is evaluated within a novel resource economy, in which physical resources, trust and money are treated homogeneously. A second-order trust model continually updates trustworthiness opinions, based on contract performance; trust delegation certificates support flexible, distributed extension of these trust relationships. The introspectible contracts, resource and trust models together provide accountability and resilience, which are particularly important for large-scale distributed computation initiatives such as the Grid. Thus participants can take calculated risks, based on expressed policies and trust, and rationally choose which contracts to perform.

1 Introduction

This paper presents a contract framework in which sophisticated accounting policies can be specified. Participants' trustworthiness is constantly measured against these policies, ensuring that resources are allocated rationally. This provides robust protection against fraud, even in widely distributed environments such as the Grid.

These introspectible contracts are integrated into a virtual economy with a sophisticated second-order trust model. In this economy, the trustworthiness of participants is constantly assessed and updated, through the use of accountable contracts and resource measurement. Thus, participants can estimate the risks of contracts, both in advance and while they are being performed.

Section 2 introduces the contract framework within which participants interact, while section 3 details the language in which participants express their contractual obligations to each other. Next, section 4 presents the resource model of the contract framework; here trustworthiness is measured as the ratio between actual and specified resource reimbursement for a contract. Trust transfer mechanisms are also proposed, for scalable distribution of these local trust measurements. In section 5, the steps involved in the execution of a contract are illustrated, and section 6 describes the testing framework in which the efficiency and robustness of the contract framework is assessed. Finally, section 7 summarizes the contributions of this work.

2 Contract Framework

The contract framework in which accounting policies are specified is introduced in this section. In this architecture, contracts are the building blocks of relationships. They codify agreements between participants, and emphasize the distinction between planning and performance of actions. Since the accounting function of each contract is separated from the contract action, a participant can predict whether a contract is suitable before accepting it, by treating the accounting function as the contract's policy statement. This introspection allows better planning of resource needs, and makes explicit the risks of dealing with other, possibly untrusted participants. Other frameworks also emphasize some of these needs: Gisler et al. [11] outline a secure process for electronic contracts, while Linington, Milosevic and Raymond [18] stress the importance of applying trust metrics to contracts.

This section details how contracts themselves are represented, and the contract negotiation protocol by which they are established.

2.1 Contract Representation

A contract represents an action to be performed, and simultaneously provides a high-level description of the action. In our framework, this layering is made explicit in the following attributes of contracts:

Server identity. This identifies the participant that will perform the contract action.

Client identity. The client is responsible for reimbursing the server for resources used in performing the contract.

Estimated resource consumption. This has three constituents: constant consumption, rate of consumption, and estimated maximum durations. It is the highest level description of the contract, and allows approximate resource allocation and planning, moderated by trustworthiness considerations.

- **Constant consumption.** A collection of basic resources, summarizing the resource inputs needed to perform the data-driven [14] portion of the contract.
- **Rate of consumption.** Similar to constant consumption, but detailing the expected resource consumption per second, for the time sensitive portion of the contract; this could ensure the promise of resources for multimedia applications, for example.
- **Maximum durations.** These identify the maximum and expected durations of the constant and time sensitive processing.

The overall estimated resource consumption would then be (constant consumption) $+$ (rate of consumption) $\times$ (expected duration).

Accounting function. The accounting function describes the contract's accounting policies, and determines the instantaneous resource exchange rates during the performance of the contract. This uses a limited language, described in detail in section 3, which allows only computations which will complete in a predictable amount of time — the language lacks constructs for recursion and iteration, but allows the preservation of a predefined, finite amount of state.

Contract action. This is the primary action which the server is contracted to perform. Its format is independent of the contracting architecture, but it would typically identify a program or subroutine to be executed. For example, this might be a Java method signature, specifying an action to be performed in a secure sandbox [12].

Terms of payment. The time allowed for payment after the use of a resource is specified here. If resources are required as a pre-payment or deposit, this can be specified in the accounting function.

Contracts must be signed by both client and server to be valid or enforceable; the signing and security of contracts is discussed in more detail in 2.2 below. The distributed trust framework is then used to assess which contracts are to be accepted — contracts themselves do not usually make trust issues explicit. However, trustworthiness can be used to determine which contracts are offered to particular clients, providing better terms for trusted participants.

Resilience and accountability are particularly important for applications such as peer-to-peer 'compute servers' and the Grid [19] — the concept of creating a global distributed computing framework as ubiquitous and usable as the electricity grid. One of the great challenges of the Grid will be enabling unfamiliar services to interact in novel and meaningful ways. For example, a ray-tracing service idle at night in Japan might render a video sequence for a design company in America, while it was day-time there. Only by contractually formalizing agreement on actions to be performed, and taking fluctuating trustworthiness into account, can these services be made reliable and self-supporting.

2.2 Contract Protocol

Contract acceptance requires a contract negotiation protocol with sufficient security to discourage attacks and support trust management.

The contract protocol described here is a point-to-point protocol for non-repudiable contract negotiation, performance and termination, using typed asynchronous messages between participants. The tasks of providing directory services and initially matching suitable servers and clients are considered prerequisites, which can be provided by techniques such as the contract net framework [21].

We assume that a secure public-key framework for digital signatures is available, such as PKI [8], in which participants sign messages to prove that they accept them. It must be computationally infeasible to fake the signature of a given message, or to find another message corresponding to a known signature. Furthermore, if a signature is compromised, then it loses all trustworthiness and its owner will have to create a new identity to enter into new contracts, with a concomitant loss of trust.

The contract protocol introduces a new accounting message type to the usual message types in the asynchronous messaging system. There are three basic sub-categories of these accounting messages:

Contract A contract message represents a pact between a client and a server: the server will perform an action on the client's behalf, for which resources will be exchanged.

Payment A payment message identifies a contract, and includes resource tokens such as digital money or a digital cheque owed in terms of the contract.

Termination A termination message ends a contract. This may occur once all services and payments have been completed, or it may signal premature termination by client or server. Clients and servers recovering from a crash should send termination messages for all contracts they can no longer support to ensure closure without further loss of trust.

Signing, countersigning and requests are combined with these messages, to complete the accounting framework:

Signed messages prove that the signatory will be bound by the message contents. Contracts, payments and terminations must be signed to be binding. Ordinary messages may also be signed, if extra security is required.

Countersigned messages prove that the countersigner has received the signed message, and also agrees to its contents. Contracts, payments and terminations may be countersigned. Ordinary messages are countersigned only if proof of receipt is required.

Request messages wrap ordinary accounting messages, and ask the other party to send a contract, payment or termination. Request messages are typically signed as well. (This prevents a third party from staging denial of service attacks by generating fraudulent requests.) Other ordinary messages are generally not wrapped in requests.

It is assumed that messages from each participant are uniquely identified by a local timestamp or sequence number. Stale or replayed messages should always be ignored. (Messages also include an expiry time or expiry sequence number. A stale message is one received after its expiry time, or after another message from the same source which postdates the expiry sequence number.)

In summary, the contract description language and protocol enable participants to exchange services by negotiation. Contract descriptions have many levels, allowing accurate prediction and accounting of the resource costs involved. As we will show, by combining these predictions with trust histories, participants can make rational decisions about which contracts to accept, and monitor the contracts as they are performed. Since all resources are accounted for, attacks such as denial of service are thus more easily detected and prevented. By using a language with a constrained syntax for accounting functions, accounting overheads and utility can be predicted. This allows complex contract terms to be specified even by untrusted clients, and analysed to predict the risks, even before they are executed within the contract architecture.

3 Accounting Language

Our contract framework uses a novel accounting language, to specify the resource exchange rates for each contract. This allows complex accounting policies to be specified, including those based on market prices or featuring sophisticated pricing functions.

Accounting functions are written using a simple procedural language, to represent the exchange of resources required by a contract. The grammar of this language has been designed to ensure that all accounting functions are also legal statements in the Python language [24], so that it may easily be learnt. However, its expressive power is deliberately limited, to guarantee that accounting operations can be performed in a predictably short time interval. An example of an accounting function follows:

```
class Accountant(resources.ResourceContract):
   importList = ['localResourceValues1']
   totalCPU = 0
   def processResourceAtom(self, atom, imports):
      if atom.type != resources.cpuTime:
         return [] # Charge for CPU only
      rate = imports[0]
      if self.totalCPU < 10: result = rate+0.01
      else:                  result = rate+0.002
      self.totalCPU += atom.quantity
      return [ResourceAtom(resources.money, '£',
                           result*atom.quantity,
                           atom.startTime,
                           atom.endTime) ]
```

This illustrates a sophisticated payment policy, in which a contract action will be charged for the CPU time which it uses. The price proposed is slightly more than the market rate, but with a discount if more than 10 seconds of CPU time is used.

No direct communication between the accounting function and the contract action is allowed — this ensures that the accounting code is entirely self-contained. This also allows servers to simulate and predict the effects of accepting a particular contract, and thus model the attendant risks.

The accounting language has a tightly constrained syntax. It allows each accountant to specify a list of resource rate inputs, and initial values for any persistent variables. Whenever the accountant processes a basic resource atom, it returns the list of resources owed under the contract — to do this, it can consider only its current state, the details of the atom, and the current values of the resource rates initially requested.

Two special resource atoms, with types `resources.begin` and `resources.end`, are used to signal to the accountant when a contract begins and ends, allowing initial and final charges to be implemented. For example, an accounting function might specify that the client would be reimbursed if the server ended the contract prematurely, as shown by the subtype of the terminating resource.

The state of an accountant is stored in the persistent variables listed at the start of its specification. These and all local variables may store only numbers — either integers or floating point numbers, depending on the calculation which generated them. Strings and arrays may be used only to specify imported resources and to construct new resources owed under a contract.

3.1 Accounting Language Grammar

An extract of the accounting language's BNF grammar is given below. The remaining rules are simplifications of the rules of the standard Python grammar [24], to allow only variable assignment, numerical arithmetic, and `if ... elif ... else` statements. There are no general rules for exception handling, object creation, array or list processing, iteration or function calls; instead, specific rules and keywords allow resource atoms to be returned and import lists to be specified.

```
accounting_input: 'class' NAME '(' BASE-
   CLASS ')' ':'
    NEWLINE INDENT import_list var_list+
    accounting_fn DEDENT
import_list: 'importList' '=' '[' [string_list] ']'
    NEWLINE
var_list: NAME '=' NUMBER NEWLINE
accounting_fn: 'def' PRO-
   CESSFN '(' 'self' ',' 'atom'
    ',' 'imports' ')' ':' suite
suite: simple_stmt | NEWLINE INDENT stmt+ DEDENT
stmt: simple_stmt | if_stmt | return_stmt
return_stmt: 'return' '[' [result_list] ']'
result_list: resource (',' resource)* [',']
simple_stmt: small_stmt (';' small_stmt)* [';']
    NEWLINE
small_stmt: expr_stmt | pass_stmt
expr_stmt: test (augassign test | ('=' test)*)
test: and_test ('or' and_test)*
not_test: 'not' not_test | comparison
...
resource: 'ResourceAtom' '(' test ','
    (test | STRING) ',' test ',' test ',' test ')'
atom: '(' [test] ')' | NAME | NUMBER |
    'imports' '[' NUMBER ']'
```

Additional checks on accounting functions are performed when they are compiled, such as ensuring that only previously declared persistent variables are referenced. The lexical analyser also imposes certain restrictions, particularly on the use of dotted notation: all occurrences of `self.var` are parsed into a `NAME` token, where `var` is any legal variable name. Similarly, `resources.id` and `atom.id` are `NUMBER` tokens, provided that `id` is one of the predefined attributes allowed.

3.2 Special Considerations

Flawed accounting functions may cause run-time errors, such as arithmetic errors or out of bounds indexing of imports. In that case, the contract server's actions are unspecified. Depending on the circumstances, it would be reasonable to ignore the error or cancel the contract (and decrease the client's trustworthiness rating), or to estimate the reimbursement based on previous values. The contract's client may also be notified, so that the error can be corrected in future contracts.

The granularity and ordering of resource atoms can also affect pricing. For example in the accounting function given above, different prices would have been assigned to 20 seconds of CPU usage, depending whether these were presented to the accountant as a single 20-second entry, or as two 10-second entries. Changing the order in which resources are presented to an accountant can have a similar effect. The granularity issue can often be resolved by writing the accounting function more carefully. For example, the code in lines 8-14 above could be replaced with

```
newCPU = self.totalCPU + atom.quantity
if newCPU < 10:
    cost = (rate+0.01) * atom.quantity
elif self.totalCPU<10 and newCPU >= 10:
    cost = (rate+0.01) * (10-self.totalCPU) \
         + (rate+0.002) * (newCPU-10)
else: cost = (rate+0.002) * atom.quantity
self.totalCPU = newCPU
return [ResourceAtom(resources.money, '£',
                     cost,
                     atom.startTime,
                     atom.endTime) ]
```

The contract framework guarantees that events will be presented to the accounting function in increasing order of start time, which greatly reduces order-related fluctuations. Beyond that, it is the responsibility of accounting function authors to ensure that reordering causes only small price fluctuations.

3.3 Predictability and Stability of Accounting

Because the accounting language has no looping or recursive constructs, each statement in a specification will be executed at most once per atom processed. Only simple operations are allowed, so each statement can be guaranteed to complete within a predictable amount of time. Therefore the total resources required per atom by the accountant can be predicted. Thus the accounting overheads of a contract action are limited by the number of resource atoms generated. (Resource atoms are defined later, in section 4.)

Accounting overheads result in additional resource consumption, generating more accounting overheads. However, these overheads can be controlled and minimized by adjusting the resource atom size used by the server, preserving system stability. When a contract terminates, there may be one accounting iteration unaccounted for. Nevertheless, this overhead is both small and bounded, and can therefore be factored into resource predictions.

Servers and clients can thus predict the suitability of a contract, and ensure that it will be profitable, by analysing the terms of the accounting function and predicting its overheads. Many contracts will be profitable over all circumstances, moderated by client and server trustworthiness. Otherwise, an estimate of the cost of stochastically simulating the contract may be obtained easily from the contract's estimated resource consumption. If this cost is justified, then the simulation will allow the profitability of the contract to be estimated under the expected situations. From this, a participant can decide whether to accept it.

Accounting policies allow participants in the contract framework to prioritise their actions, and plan their behaviour. The following section outlines a virtual resource economy and trust model, for enforcing these contracts. Section 5 then shows how this overarching trust policy may be implemented, while section 6 presents tests to assess the effectiveness of these policies.

4 Resource and Trust Models

Contractual obligations must be properly enforced, to ensure the robustness of the contract framework presented here. This requires a resource model in which all resource usage is measured, and also a trust model to identify and discourage cheating. Here, trust distribution can improve efficiency, by allowing participants to vouch for others' trustworthiness by standing surety. This is particularly important for widely distributed systems, where centralised trust management or reputation schemes are inappropriate, and where individual participants' keys are occasionally compromised by hackers. This section outlines our resource and trust models, and introduces trust delegation certificates.

4.1 Resource Model

For risk analyses to be realistic, a local account must be kept of all of the resources used, both while planning and also retrospectively. By comparing these values, current and completed contracts can be monitored to identify breaches of trust and incorrect budgeting. This helps prevent losses and avoid both accidental and malicious exploits of contract design. Operating system architectures which give applications explicit control over their resource usage, such as Nemesis [20] and the MIT exokernel project [16] would greatly facilitate this, though conventional operating systems could also be used.

All resources must be measured and incorporated, including those used in establishing the contracts; in this context, money can be seen as just another scarce resource, but with the advantage that it is fungible and commonly exchanged for other resources. Similarly, trustworthiness can also be seen as a resource, albeit subjective and difficult to measure.

The actual exchange of resources is dictated by contracts here; these may be facilitated by the addition of capital — by providing proof of trustworthiness or solvency, the contract may proceed more easily [6]. In the contracting architecture, each basic resource has five attributes:

Resource type. This distinguishes fundamentally different resources from each other, such as CPU cycles, network bandwidth, disk storage, money and trust.

Subtype. Identifies the location or denomination of the resource, e.g. the computer name for CPU usage, or network type and destination class for bandwidth, or currency for money.

Quantity. The number of units of the resource used, or expected to be used.

Start time, and

End time. These mark the time during which the resource was to be used.

It is assumed that the resource was used evenly over the associated time interval. Otherwise, the resource usage should be represented using shorter intervals.

This resource representation is a compromise between expressiveness and compactness. It allows arbitrarily fine-grained accounting where necessary, but can also be used for summary representations of large quantities of resources. For most real applications, minute precision in accounting is unjustified, since the accounting overheads would overwhelm any savings. The precise break-even point will vary between systems and between contracts; furthermore, external practical or accounting considerations may constrain this choice.

Resources can be combined as well as split; assuming two basic resources have the same type and subtype, and adjoining or overlapping time intervals, then they may be combined into a larger basic resource — the maximum discrepancy allowed here is an accounting decision. The limits of amalgamation show the fundamental resource types in this framework; these are represented as pairs consisting of resource type and subtype.

The value of a resource depends on which other resources it can be exchanged for. The contract architecture prices resources by establishing the terms of exchange within each contract. Participants then accept contracts only if they expect that the rewards of the contract will exceed the costs of the resources which they contribute. By assigning realistic costs to resources, and values to contracts, it becomes possible for computers to decide on the most effective use of those resources. During idle periods, for example, there would be an oversupply of computational resources, which could be offered at a discount in exchange for other resources, such as money.

Accurate resource accounting requires proper support from the underlying operating system, in allocating and accounting for resources. Nevertheless, even approximate resource accounts can be used, provided that they overestimate the real resource consumption. This is because, as long as the computer system is meeting its real-world costs, it remains profitable to maintain. A more awkward problem occurs when participants are using a shared operating system or other common resources, and are unable to predict their own resource allocations. In that case, the participants may lose trust because of the underlying unpredictability, unless they can reserve resources in advance.

Resource accounting reduces the risk of Denial of Service and other attacks. This is because any leak of resources will be represented by an unprofitable contract or other process, which will in future be given a lower priority. Therefore, these attacks are possible only by infiltrating a number of trusted participants simultaneously, or by expending more resources on the attack than the costs to the victim, which is ultimately ineffective.

The representation scheme for resources shown above is an essential requirement for the contract architecture presented in this paper. It provides a complete and homogeneous resource model for contracts to describe their needs, for both prospective and retrospective accounting. The unification of both fine- and coarse-grained representation of resources allows multi-resolution risk analyses to be performed, and accounting overheads to be controlled.

4.2 Subjective Trust Model

A subjective trust model is one of the foundations of the contract architecture. This allows participants' trust beliefs to be constantly updated during the course of their interactions. In this paper, a second order model of trustworthiness is adopted, based on Jøsang's [15] Subjective Logic, an extension of the Dempster-Shafer theory of evidence, and proposed for use in electronic markets by Daskalopulu et al. [5]. This allows uncertainty in a trustworthiness assessment to be considered explicitly, in contrast with simpler trust representations.

Conventional trust systems assume that trustworthiness is known in advance, and unchanging. The degree of trust is then represented in various ways: as a number in economic models [4], as membership of a trust class in privacy systems such as PGP [3] and in other trust frameworks [1], or as membership of a role in access control systems [2]. These models implicitly assume that there will be no further information about agents' trustworthiness, and therefore do not represent the accuracy of the knowledge assessments.

Trust models are frequently designed for security applications, which must ultimately make a once-off decision to accept or reject a user's credentials based on the trustworthiness estimate. Thus, no further provision is made for limiting the risk of fraud from authenticated participants, since these conditions are very difficult to express as security policies.

By contrast, the need to take calculated risks is a cornerstone of a contract performance framework. For these decisions to be sensible, constant reassessment of trustworthiness is required to discourage cheating and encourage cooperative behaviour. In the game-theoretical Prisoner's Dilemma, simple retaliatory automata perform well, though a degree of forgiveness is required when measurement errors may occur [13, 17]. However more complex introspection is required when prioritising resource allocations, in order to identify the most productive contracts — the classic Prisoner's Dilemma does not allow the selection of opponent.

In subjective logic, a combination of belief, disbelief and uncertainty functions represents the apparent trustworthiness of a participant. These values are subjectively determined by each participant, based on their experiences. For example, if participant A knew nothing about participant B, then A would initially assign a belief value of 0, a disbelief value of 0, and an uncertainty of 1 to proposition φ that B would behave truthfully. Thus A's opinion ω_{φ}^{A} would be represented by the triple $(0, 0, 1)$. Conversely, if A knew that participant C had been truthful in only 5 out of 10 dealings, then A might hold the opinion $\omega_{\psi}^{A} = (0.3, 0.3, 0.4)$ where ψ is the proposition that C would behave truthfully. One of the coordinates of each opinion triple is redundant; their sum is always 1.

Strictly, a fourth value should be included: the relative atomicity, which measures the overlap or correlation between the data on which the opinion is based, and the domain of the proposition φ under consideration. This relative atomicity is required to accurately estimate the expected probability of φ:

$$\mathrm{E}(\varphi) = b(\varphi) + a(\varphi)u(\varphi) \text{ where } \omega_{\varphi} = (b(\varphi), d(\varphi), u(\varphi), a(\varphi)) \tag{1}$$

In this paper, for simplicity, it is assumed that past behaviour is a good predictor of future behaviour, i.e.

$$\mathrm{E}(\varphi) = \frac{b(\varphi)}{b(\varphi) + d(\varphi)} \text{ if } b(\varphi) + d(\varphi) \neq 0, \text{ otherwise } \mathrm{E}(\varphi) = k \tag{2}$$

where k is a constant reflecting the expected behaviour of previously unknown participants.

By making uncertainty in trust explicit, it is possible to estimate the effects of decisions based on trust, and their expected bounds. In the above example, given $k = 0.5$, A's expected returns would be the same when transacting with B or C; however, the predicted minimum and maximum returns would cover a wider range for B than for C. This would be particularly important if the cost of a failed transaction were significantly greater than the benefits of a successful transaction, or A were very risk averse.

Because this framework is subjective and based on experience, each participant forms its own opinions of others' trustworthiness. The mechanisms for this are outlined in the next section, while the following section shows how participants share their opinions to provide more accurate trustworthiness estimates.

4.3 Trustworthiness Measurement

Here we show the application of subjective logic to contractual agreements between computerised participants, within our contracting framework.

Opinions in the subjective logic are based upon belief mass assignments; these reflect the apparent probabilities of state combinations within the appropriate frame of discernment. In our application, each frame of discernment represents the trustworthiness of a participant, ranging continuously between 0 (always cheats) and 1 (always reliable). These belief masses are assigned according to each participant's contract performance.

A contract is an agreement between two or more parties about the actions they are to perform; participants enter into contracts which they expect to be to their advantage. In this paper, contracts involve an exchange of resources, and participants aim to obtain resources which they consider valuable. Progress is achieved through the constant production of resources by hardware and users, which are then consumed in the performance of contracts.

Opinions are formed based on completed actions, and the corresponding belief masses are weighted in proportion to the return on investment of the actions. These completed actions may represent entire or partial contracts, provided only that their success can be assessed. Thus for each completed action, the belief masses represent the expected truthfulness of the corresponding participant. For example, if a transaction were successfully performed, the continuous belief mass function $m_\varphi(x) = 2x$ might apply, where $x \in [0, 1]$. This represents no belief that the participant always cheats, and a high belief that the participant is truthful.

The individual belief mass functions of each action are then combined according to their resource commitments, and normalised, to yield the overall belief mass function for a participant.

This framework allows continual updating and refinement of trust estimates, based on actual behaviour. This is essential in ensuring that best use is made of the available resources. In addition, it provides protection against participants whose security has been compromised, and against a participant engaging in numerous small transactions with a view to cheating on a large transaction later.

If a participant is compromised, an intruder might gain control of its signature and use this to steal resources under an assumed identity; the trust framework allows participants to limit this risk, and detect resource leaks (unless the intruder can also fake network packets).

Because trustworthiness is based on the value of resources, not the number of transactions performed, participants cannot generate trust spuriously. Furthermore, trustworthiness is continuously monitored, so only very limited resources will be speculated on participants that have not proven their trustworthiness, or on participants that begin to cheat.

4.4 Transfer of Trust

The subjective logic framework is well adapted to large systems, since it is not dependent on a central trust or reputation authority. This allows good scalability, because each participant will have dealings with only a relatively small number of others.

However, the disadvantage of this occurs if contracts are usually between participants previously unknown to each other. The corresponding lack of trust could stifle the formation of new contracts, particularly if unknown participants were generally unreliable — thus even prolific and completely honest participants might find it impossible to establish contracts. In effect, all participants would be punished for the untrustworthiness of a few.

These difficulties can be overcome in two ways: by pooling trust information, or through trust delegation mechanisms together with only local trust.

If participants pool their information, their trust estimates can be improved, but only if the contributing participants are trustworthy themselves and not supplying disinformation. One solution would be to share trust information through the equivalent of a credit bureau. If all credit entries at the bureau included evidence, such as signed contracts, the risk of slander would be limited to those participants that had previously dealt with each other. However, this solution relies on continuity of identity for its effectiveness, increasing the costs of remaining anonymous. (It may also require post-unforgeable transaction logs, to protect the bureau's entries.)

This pooling of trust would therefore be most appropriate within an organisation or institution, where fraud and reliability issues could be most easily managed. Nevertheless, conventional security and logging mechanisms would still be needed to prevent an attacker from infiltrating one of the participating computers and thus corrupting the trust repository. (With careful design, the effect of a single intrusion could be limited, but distributed intrusions would remain an issue.) Other techniques for reducing slander are discussed by Dellarocas [7], while Szabo [23] outlines the general limitations of reputation systems.

In this context, the attack could be a worm installed on each of the participating computers, intercepting and modifying messages to the credit bureau before they were signed. The effects of the intrusion could be reversed, but detecting the intrusion would be very difficult. If the attack were limited to a single computer, this could be detected statistically from discrepancies between its credit bureau entries and those from other contributors. By requiring consensus for all credit bureau recommendations, the results of the attack could even be hidden from the users, but at the cost of ignoring potentially useful data.

An alternative is to allow participants to delegate trust to each other, through the issue of certificates. These trust delegation certificates bind the trustworthiness of pairs of participants together. For example, participant A might issue a certificate of 50% trust to participant B, stating that she is prepared to pay 50% of B's bad debts. Therefore participants that trusted A would increase their trust in B, by incorporating A's trustworthiness.

PGP has an analogous mechanism whereby one participant can sign another's public key, to act as an 'introducer' to a third party. The resulting 'web of trust' [3] allows the identity of previously unknown participants to be verified indirectly, to allow secure communication to take place. This was novel because it allowed arbitrary trust relationships, instead of enforcing hierarchical delegation of trust. Similarly in our framework, it is possible to build complex chains of trust, either systematically or in an ad hoc manner.

There are four main constituents to a trust delegation certificate:

Guarantor. A participant willing to accept responsibility for making reparation, should the guaranteed party default on a contract.

Guaranteed party. A participant that is partly trusted by the guarantor.

Terms. These typically specify the percentage of the debt that would be repaid, the maximum resource reimbursement, and the domain in which the certificate applies.

Signature. The guarantor's public key is used to prove the authenticity of the trust certificate.

When a participant receives a trust certificate, it uses the discounting operator of subjective logic to adjust its opinion of the guaranteed party's trustworthiness. This operator allows one participant to incorporate another's advice about a proposition.

The use of these trust certificates extends beyond standing surety for a close acquaintance. They could also be used on a larger scale, to create communal entities for trust sharing — the equivalent of a cooperative association for bulk purchases. Similarly, companies could issue certificates to employees, allowing them to commission services as company representatives.

In summary, a sophisticated trust model is an essential constituent of a distributed contract architecture. It allows trustworthiness estimates to be continually updated, incorporating new data and recommendations. This allows participants to better allocate their resources, and assess the risks of contracts under consideration. These trust estimates are local and subjective — and thus encapsulate the basic trustworthiness of the other participant, and the reliability of both participants' environments and the messaging service between them. With trust delegation, participants can stand surety for each other, cooperating to make best use of their trust.

5 Implementation

Each server has a scheduler to guide the execution of its contracts. It selects and processes contracts with the help of trust recommendations, while recording all resource usage. It includes a communication module which is responsible for collecting and distributing the reimbursements required by contracts, and for resending signed messages which might have been lost in transit (in the case of unreliable asynchronous messaging).

For a cluster of servers, a proxy scheduler could be used to pre-accept contracts on behalf of the cluster. Correspondingly, each client has a user agent to request and accept contracts, and make appropriate payments on the client's behalf; this agent would typically seek the user's permission before accepting a contract, but make payments automatically, by weighing the risk of a mistaken payment against the cost of the user's time to make the payment decision.

At the heart of the scheduler is a list of current contracts. Each contract has the following associated information:

Status. This determines whether the contract is current or not, and whether or not it should be processed. The states are similar to those of an operating system scheduler [22], but include additional states for contract negotiation and termination.

- **Server proposed.** The server has offered a signed contract to the client. If the client returns a countersigned copy within the given timeout period, the contract will come into force. (There is currently no corresponding

'Client proposed' state, since this would last only momentarily while the scheduler decided whether or not to accept it.)

- **Ready.** The contract has been signed and countersigned by server and client, and is ready to be executed.
- **Waiting.** The contract is waiting for input, or for a particular time (for continuous multimedia applications).
- **Running.** The contract is currently being executed.
- **Server terminating.** The server has signed a contract termination message, and is waiting for the client to countersign it.
- **Client terminating.** The client has signed a termination message, and the server is waiting for outstanding payments to be collected.
- **Terminated.** The contract termination has been signed and countersigned.

Contract terms. The specification of this contract, outlined in section 2.

Resources allocated and used. Resources set aside for this contract, to avoid scheduling more contracts than can be processed, and the allocation used already.

Total resources used to date. This has two parts: resources used and accounted for (including reimbursements to the client), and fractional resources not yet accounted for. The reimbursements should be signed and countersigned.

Total reimbursement received. This is a list of reimbursements, with signatures.

Cost of reimbursable resources. The cost to the system of the resources which the contract has used, which have either been reimbursed, or for which the reimbursement period has elapsed even though they have not been reimbursed.

Contract profitability. This is the ratio of reimbursement received to reimbursable resources, and is used to update the trustworthiness of the contract client.

Signatures. A list of signatures, which may include client and server signatures of the contract, and client and server contract termination signatures.

At each scheduling opportunity, the scheduler returns waiting processes to the ready queue if their conditions have been satisfied. It then selects a process to run from the ready queue, giving precedence to those processes for which (Resource allocation unused $\times$ Contract profitability) is highest. This may be seen as a variant of the stride scheduling algorithm [25] for operating systems. If all resource allocations have been used, then contracts are prioritised by profitability.

The selected process then runs until it terminates or is interrupted — because one of the waiting processes is ready to proceed, or because it has exhausted its current resource allocation.

The process's accounting function is then passed any resources used, to determine the reimbursement required. The contract metrics and client trustworthiness are updated accordingly, and the resources used for this and for the accounting are themselves added to the list of resources to be accounted for on the next iteration.

Finally, any new communication required is performed. This includes receiving contract requests and deciding whether or not to sign them, based on the current resource allocation and their expected profitability. Communication resources are charged to the corresponding contract. Any resources used in the scheduling iteration which have not yet been accounted for are assigned to a special null contract, to ensure that no resource leaks go undetected; only if the null contract exceeds its predetermined allocation is a warning produced. The scheduling cycle then repeats itself.

Simple resource reservation is used in the current model, to ensure that the server does not over-commit itself and therefore earn distrust. A consequence of this is that the server will not discard a current contract simply because an apparently more profitable contract is presented. Two parameters allow this behaviour to be adjusted, instructing the server to over-commit itself only by a certain percentage, and specifying the profitability ratio at which a contract will be spontaneously discarded. In this way, contracts which are proving unprofitable can be terminated prematurely; this would occur only if a client proved less trustworthy than initially expected, otherwise the contract would not originally have been accepted.

For honest clients, this resource reservation model provides consistency — once a contract has been accepted it will be performed, because the server attempts to remain trustworthy. Otherwise, if a server decided to give a new, more profitable contract precedence over an existing contract, it would also have to estimate the secondary costs of the resulting loss of trust.

The contract execution engine presented in this section prioritises the execution of contracts according to profitability. It uses the trust model to help calculate expected returns, and then uses resource measurements as feedback to update its opinions of trustworthiness. In this way, it aims to make best use of its resources while taking cost and trustworthiness into account.

6 Testing Framework

To assess the effectiveness of the contract framework described in this paper, automatic tests are required. Thus, introspectible contracts and second order trust techniques can be compared against simpler solutions. This section outlines an automatic testing framework, to measure the performance of our architecture.

The tests are performed using computers on a local network, but delays and unreliability are artificially introduced to simulate performance in a widely distributed network such as the Internet. Although large networks are difficult to simulate [26, 9], each participant in our contract framework interacts with only a limited number of other participants over a given interval. As a result, we need to simulate only a small group of nodes — though these may be widely dispersed in the underlying network topology. The interactions of this group with the rest of the network can be modelled iteratively in terms of aggregated resource overheads, provided that performance is measured for only the internal nodes of the group.

6.1 Network Simulation

The contract framework is implemented using asynchronous message-passing semantics, detailed in section 2. Therefore we simulate network behaviour at the message level in the testing framework. Because only relatively small networks with hundreds of nodes are simulated, we make the following simplifying assumptions:

1. The topology of the simulated network is specified explicitly in advance. That specification could be generated automatically from a statistical model, or based on measurements from an actual network. The topology could also be changed on the fly within our architecture, though this has not yet been implemented.

2. Each simulated node has a routing table, which determines the route each message will follow over the underlying network — either directly to its destination, or via an intermediate node. This allows shared bandwidth network links to be simulated. The intermediate node may or may not be one of those visible to the contract framework; for example a 128kbps ISDN bridge from a corporate network to the Internet could be simulated by adding a concealed node, and routing all non-local messages through it.

3. The underlying, simulated network links are represented by a table showing bandwidth and reliability for each pair of nodes. For efficiency in networks with sparse links, this is implemented with a hash table, with no entries for node pairs which are not directly connected in the routing tables. Initially, bandwidth and reliability are represented as a pair of numbers, but accurately modelling extra congestion overheads or congestion collapse [10] on networks such as non-switched Ethernet would require a more advanced representation. A lower bound for performance can be obtained by assuming that the worst-case congested bandwidth is available.

4. Each simulated node is also given a list of the CPU, network and storage resources at its disposal, and a reliability description: mean time to failure, failure mode and recovery interval. This allows modem users and slow clients to be simulated, and also client failures. A failed node also does not forward others' messages, until it recovers.

Timing issues can be important if the message simulation overheads are higher than the delays expected in the simulated network. Therefore, the message simulation may need to slow the actual performance of contracts proportionally to ensure that the simulation remains faithful. The current implementation achieves this by detecting messages received after they were supposed to be delivered. (All simulating nodes are assumed to be on a local network, so that the clocks can be synchronized, or clock skews can be measured and accounted for.) The messaging subsystem then instructs the execution engine's scheduler to insert artificial delays in local contract execution. This allows each local simulated clock to lag behind the real time. The execution engine's scheduler also limits the CPU resources used locally to the budget given when it was created.

A later implementation will make the simulation framework more generally applicable by introducing both forms of lag by temporarily blocking on calls to the message passing simulator instead, or pausing processes using the operating system scheduler. These may be seen as ways of constraining resource usage, suggesting that the contract framework could eventually be used to simulate itself.

Clock adjustments are propagated through the simulation network primarily when other messages are sent. Each conventional message is annotated with the real time at which it was sent, the simulated sending time, the lag factor on the sending node, and the sender's identity. If a message is received at a simulated time that significantly precedes the simulated sending time, that message is queued before being delivered locally; it may also be delayed further because of simulated network delays. In addition, the sender should temporarily pause its local simulation, allowing the recipient to resynchronize its time frame. Clock lag factors propagate in the same way, to minimise the number of corrections required.

To avoid large clock skews between independent network partitions, occasional keep-alive messages are sent within the simulation network. These messages also allow clock lag factors to be reduced: if no messages are received too late to deliver (after taking into account simulated network delays), then the lag factors may be too high.

Each real machine may simulate a number of nodes. In the current implementation, each node simulator runs as an independent process, operating a single contract scheduler. For greater efficiency, a shared library could be used instead, reducing message passing overheads and allowing lag factors to be coordinated more efficiently.

6.2 Initialising the Simulation

The testing framework is initialised using a script, combined with the topology specification described above. A control module executes the script and also acts as a well-known node in the simulation network, allowing it to interject contracts and other messages, and to establish initial trust relationships for the simulation. The script therefore specifies a list of initial contracts which are automatically accepted and interpreted by the control node; these can spawn other contracts within the simulated network. Each of these contracts also has an associated start time, and a list of node and subjective trustworthiness pairs; this trust can then be transferred to participants on other nodes in the simulation via trust delegation. Finally, the control node passes the complete list of simulated nodes to each initial contract action clause, allowing contracts to be distributed throughout the network.

6.3 Tests and Results

Testing of the contract framework aims to measure the overhead it introduces, and the effects of introspection and constant trust assessment. The results are collated by contracts themselves, and also by the network simulation environment. Three tests were carried out:

Contract Overheads. This tests stochastic optimisation with and without contracts. In the contract-based portion of this test, the control node establishes a PBIL optimisation server at one node, which uses contracts to establish clients on remote nodes. Each client runs optimisations for approximately a minute (of simulated time), returning intermediate results at 10 second intervals. The server spawns 10 generations of client before reporting its conclusions to the control node.

The conventional portion of this test uses dedicated PBIL clients, with the same simulated network topology: ISDN links between all nodes. The results of this test give the CPU and network overheads of the contract framework.

For this test, contract messages accounted for 26% of all communication messages, but only 16% of the communication bandwidth. The CPU overheads were approximately 2% of total CPU usage.

Introspection. This test assesses the usefulness of introspectible contract accounting. For this test, a server with or without introspection is presented with five contracts within its resource budget. The first proves unprofitable to perform (though this may not be clear in advance), two are exceptionally lucrative, and the others are moderately profitable. When a contract is completed (each uses a minute of simulated CPU time), another identical contract is offered to the server; after 10 minutes, the overall profitability of the server is measured, to determine whether it has prioritised effectively.

Using the conservative resource allocation strategy described earlier, the simple server treated all contracts equally, and had an overall profitability of 26%. With introspection, the profitable contracts were successfully identified, and the overall profitability rose to 38%.

Trustworthiness. This test compares second order trust modelling and blind trust. Here, four clients request one contract each on a single server. The first client is always trustworthy, the second pays 50% of its bills, the third pays diminishing fractions of each bill, and the last pays each bill late. The server's costs and the contract terms ensure that at least 30% payment is required for a contract to be worthwhile. Each contract uses 1 minute of simulated CPU time, and each contract is renewed when it terminates. Again, the overall server profitability is measured, for each of the trust models.

This test yielded profitability values of 165% and 132%, although further tests are still needed, with more complex clients in a larger network.

7 Conclusion

The contract framework presented in this paper allows flexible contracts to be established between computers, based on a novel language for accounting policy. These policies are interpreted within a virtual resource economy in which favourable contracts are identified, moderated by trust assessments; participants take calculated risks, then audit the results to update their assessments of trustworthiness. The homogeneous treatment of trust, money and other resources simplifies the structure of this resource economy.

The accounting language has a predictable and stable execution profile, yet it allows sophisticated policies to be specified. Combined with contract resource estimates, this enables flexible prediction of contract costs, further reducing the risk of contract establishment. An execution engine schedules both conventional and multimedia tasks using a resource allocation scheme, prioritising tasks based on cost and on compliance with their accounting policies. Finally, a testing framework and network simulation model were introduced, to quantify the effectiveness of our contract architecture.

Distributed computational services are becoming increasing important for the Grid and for web services. The contract architecture presented here uses accounting policies as a significant step towards this goal; it allows a distributed network of participants to reason introspectively about resources and trust, in order to negotiate rational contracts.

Acknowledgment

We gratefully acknowledge ICL, now part of Fujitsu, for supporting Brian Shand's PhD research through the Computer Laboratory's ICL studentship.

References

[1] A. Abdul-Rahman and S. Hailes. Supporting trust in virtual communities. In *Hawaii International Conference on System Sciences 33*, pp. 1769–1777, Jan. 2000.

[2] J. Bacon, K. Moody, and W. Yao. Access control and trust in the use of widely distributed services. In *Proceedings Middleware 2001, Lecture Notes in Computer Science 2218*, pp. 295–310, 2001.

[3] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pp. 164–173, May 1996.

[4] S. Brainov and T. Sandholm. Contracting with an uncertain level of trust. In *Proceedings of the first ACM conference on Electronic commerce*, pp. 15–21, Denver, CO USA, Nov. 1999.

[5] A. Daskalopulu, T. Dimitrakos, and T. Maibaum. E-contract fulfilment and agents' attitudes. In *ERCIM WG E-Commerce Workshop on The Role of Trust in e-Business*, Zurich, Oct. 2001.

[6] H. de Soto. *The Mystery of Capital: Why Capitalism Triumphs in the West and Fails Everywhere Else*. Bantam Press, London, 2000.

[7] C. Dellarocas. Immunizing online reputation reporting systems against unfair ratings and discriminatory behavior. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pp. 150–157, Minneapolis, MN, Oct. 2000.

[8] C. Ellison and B. Schneier. Ten risks of PKI: What you're not being told about public key infrastructure. *Computer Security Journal*, 16(1):1–7, 2000.

[9] M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *SIGCOMM 1999*, pp. 251–262, 1999.

[10] P. Gevros, J. Crowcroft, P. Kirstein, and S. Bhatt. Congestion control mechanisms and the best effort service model. *IEEE Network*, 15(3):16–26, 2001.

[11] M. Gisler, H. Häuschen, Y. Jöhri, A. Meier, O. Müller, B. Schopp, and K. Stanoevska. Requirements on secure electronic contracts. Technical Report MCM-institute-1999-01, University of St Gallen, 1999.

[12] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the Java Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pp. 103–112, 1997.

[13] P. Grim. The greater generosity of the spatialized prisoners-dilemma. *Journal of Theoretical Biology*, 173(4):353–359, Apr. 1995.

[14] R. Jagannathan. Dataflow models. In E. Zomaya, editor, *Parallel and Distributed Computing Handbook*, chapter 8. McGrawHill, 1996.

[15] A. Jøsang. A logic for uncertain probabilities. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(3):279–311, June 2001.

[16] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Janotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Symposium on Operating Systems Principles*, pp. 52–65, 1997.

[17] S. T. Kuhn. Prisoner's dilemma. In E. N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Stanford University, Stanford, CA, summer 2001 edition, 2001. [Online]. Available: `http://plato.stanford.edu/archives/sum2001/entries/prisoner-dilemma/`.

[18] P. F. Linington, Z. Milosevic, and K. Raymond. Policies in communities: Extending the ODP enterprise viewpoint. In *2nd International Enterprise Distributed Object Computing Workshop*, pp. 11–22, Nov. 1998.

[19] PPARC. Internet to SuperNet — the DATAGRID project, Jan. 2001. [Online]. Available: `http://www.pparc.ac.uk/nw/supernet.asp`.

[20] D. Reed, I. Pratt, P. Menage, S. Early, and N. Stratford. Xenoservers: accountable execution of untrusted code. In *IEEE Hot Topics in Operating Systems (HotOS) VII*, pp. 136–141, Mar. 1999.

[21] T. Sandholm and V. Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *First International Conference on Multiagent Systems (ICMAS-95)*, pp. 328–335, San Fransisco, 1995.

[22] A. Silberschatz and P. B. Galvin. *Operating System Concepts*. Addison-Wesley, Reading, MA, USA, 1998.

[23] N. Szabo. Formalizing and securing relationships on public networks. *First Monday*, 2(9), 1997. [Online]. Available: `http://www.firstmonday.dk/`.

[24] G. van Rossum. Python library reference, July 2001. [Online]. Available: `http://www.python.org/doc/ref/`.

[25] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional- share resource management. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, June 1995.

[26] E. W. Zegura, K. L. Calvert, and M. J. Donahoo. A quantitative comparison of graph-based models for Internet topology. *IEEE/ACM Transactions on Networking*, 5(6):770–783, 1997.