Technical Report

UCAM-CL-TR-600
ISSN 1476-2986

Number 600

UNIVERSITY OF CAMBRIDGE
Computer Laboratory

Trust for resource control: Self-enforcing automatic rational contracts between computers

Brian Ninham Shand

August 2004

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

http://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted February 2004 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Jesus College.

This research was supported by ICL, now part of Fujitsu, through the Computer Laboratory's ICL studentship, and by the Overseas Research Students award scheme, the Cambridge Commonwealth Trust, and the SECURE EU consortium.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

http://www.cl.cam.ac.uk/TechReports/

ISSN 1476-2986

Abstract

Computer systems need to control access to their resources, in order to give precedence to urgent or important tasks. This is increasingly important in networked applications, which need to interact with other machines but may be subject to abuse unless protected from attack. To do this effectively, they need an explicit resource model, and a way to assess others' actions in terms of it. This dissertation shows how the actions can be represented using resource-based computational contracts, together with a rich trust model which monitors and enforces contract compliance.

Related research in the area has focused on individual aspects of this problem, such as resource pricing and auctions, trust modelling and reputation systems, or resource-constrained computing and resource-aware middleware. These need to be integrated into a single model, in order to provide a general framework for computing by contract.

This work explores automatic computerized contracts for negotiating and controlling resource usage in a distributed system. Contracts express the terms under which client and server promise to exchange resources, such as processor time in exchange for money, using a constrained language which can be automatically interpreted. A novel, distributed trust model is used to enforce these promises, and this also supports trust delegation through cryptographic certificates. The model is formally proved to have appropriate properties of safety and liveness, which ensure that cheats cannot systematically gain resources by deceit, and that mutually profitable contracts continue to be supported.

The contract framework has many applications, in automating distributed services and in limiting the disruptiveness of users' programs. Applications such as resource-constrained sandboxes, operating system multimedia support and automatic distribution of personal address book entries can all treat the user's time as a scarce resource, to trade off computational costs against user distraction. Similarly, commercial Grid services can prioritise computations with contracts, while a cooperative service such as distributed composite event detection can use contracts for detector placement and load balancing. Thus the contract framework provides a general purpose tool for managing distributed computation, allowing participants to take calculated risks and rationally choose which contracts to perform.

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

No part of this dissertation has already been or is being concurrently submitted for a degree or diploma or other qualification at any other university. This dissertation does not exceed sixty thousand words, including tables, footnotes and bibliography.

Publications

Aspects of the work described in this dissertation are featured in the following publications:

- Brian Shand and Jean Bacon. Policies in Accountable Contracts. In *Policy 2002: IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 80–91, Monterey, California, U.S.A., June 2002.
- Peter Pietzuch and Brian Shand. A Framework for Object-Based Event Composition in Distributed Systems. *Presented at the 12th PhDOOS Workshop (ECOOP'02)*, Malaga, Spain, June 2002.
- Brian Shand, Nathan Dimmock and Jean Bacon. Trust for Ubiquitous, Transparent Collaboration. In *First IEEE International Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 153–160, Dallas-Ft. Worth, Texas, USA, March 2003.[1]
- Peter R. Pietzuch, Brian Shand and Jean Bacon. A Framework for Event Composition in Distributed Systems. In *4th International Conference on Middleware (Middleware'03)*, Rio de Janeiro, Brazil, June 2003. LNCS 2672, Eds. M Endler and D Schmidt, pages 62–82.[2]
- Vinny Cahill, Brian Shand, Elizabeth Gray, Ciarn Bryce, Nathan Dimmock, Andrew Twigg, Jean Bacon, Colin English, Waleed Wagealla, Sotirios Terzis, Paddy Nicon, Giovanna di Marzo Serugendo, Jean-Marc Seigneur, Marco Carbone, Karl Krukow, Christian Jensen, Yong Chen, and Mogens Nielsen. Using trust for secure collaboration in uncertain environments. *IEEE Pervasive Computing*, 2(3):52–61, Jul–Sep 2003.
- Peter R. Pietzuch, Brian Shand, and Jean Bacon. Composite Event Detection as a Generic Middleware Extension. *IEEE Network*, Special Issue on Middleware Technologies for Future Communication Networks, 18(1):44–55, Jan/Feb 2004.

[1] An extended version of this paper has been accepted for publication in *Wireless Networks: Journal of Mobile Communication, Computation and Information*, December 2004.

[2] Best paper award.

Contents

Chapter 1

Introduction

When computers perform tasks, they normally do so blindly and with no real understanding of their actions. This dissertation shows how this problem can be resolved by using a realistic virtual economy to produce autonomous yet accountable computer systems.

For example, suppose Olivia is on the train, browsing the Internet using her laptop and mobile phone. She pays 0.2p/KB for her GPRS data link and her laptop battery is only 60% full, but that should be enough for her to read her email and the latest medical journal abstracts for an hour, before she arrives to give a presentation at a conference.

Her laptop now begins to download the latest operating system updates in the background, at considerable expense, making the web connection feel sluggish. At the same time, the disk drive begins a periodic integrity check, draining the battery much faster than before. Neither task appreciates the impact of its actions, so Olivia has to stop using her laptop after only half an hour.

At the same time, Sipho in Cape Town wants to generate 3D views of a house that he is designing, to show his clients. The ray tracing will take 30 minutes, but he'll have to wait an extra hour because the office computers are already busy. He knows that all of the computers in Los Angeles are idle where it is midnight, and wishes he could pay to have his job run there, but he does not have access to them.

Both of these scenarios illustrate the limitations of computer programs as inscrutable tasks, independent from their environment. This dissertation instead proposes a sophisticated *resource model*, which lets programs define their needs and monitor their resource usage, and an expressive *accounting model* for computing the value of the resources. These are combined in a contract framework, in which a computer's tasks are represented as *explicit contracts* to exchange resources. Inevitably, some contracts will not live up to their promises, so a robust *trust model* is also needed, to protect against attacks and failures from broken contracts.

This framework serves as the basis for many types of application, ranging from the desktop and Grid computing applications suggested above, to others involving direct human interaction and cooperative distributed services.

The following chapter reviews relevant literature on resource constrained computing, trust and security modelling, as well as self-organising distributed systems and Grid computing. We define the contract framework in Chapter 3; this includes a novel accounting model in which tasks can express complex accounting policies, but ensures that these policies have predictable, bounded resource needs.

Chapter 4 extends the scope of the contract model, by supporting interactions with untrustworthy or unreliable parties. It contributes a general purpose, formally proved, trust model for contracts, that offers safety against attacks without compromising honest and successful contracts. Personal recommendations between participants are also supported in the model, with the same safety guarantees, and illustrated in a compute server application.

Then we focus on the scope and limitations of the contract framework; Chapter 5 shows its use for non-computational resources such as the user's time and its integration with access control credentials to support self-financing web services. In addition, a PDA collaboration scenario shows the limitations of the model when it cannot assess resource usage automatically, and demonstrates how it could be extended for this application. Chapter 6 then explores collaborative distributed applications, in which the contract framework is used to implement load balancing for a composite event service, with trust as a model of competence. Finally, Chapter 7 shows the compute server implementation with performance results, followed by conclusions in Chapter 8.

Chapter 2

Literature Review

Computers need to monitor their environment and their interactions with others, to tailor their actions to the available resources. For example, a user of a peer-to-peer file backup service might want to store one backup copy of each of her files on the network if the copies are held by trusted nodes, but two separate copies when the recipients are untrusted.

This chapter reviews research into resource management, trust and security modelling and self-organising distributed computation, which all have bearing on the contract framework. However, existing work has focused primarily on individual aspects of the problem, and needs to be integrated into a single model in order to provide a general framework for computing by contract.

2.1 Resource Constrained Computing

Computer applications are often faced with a shortage of resources, which need to be apportioned to tasks carefully to allow them to operate effectively. These resource allocation issues are crucial to the performance of both individual computer systems and distributed computing environments.

Within a single computer, a CPU scheduler decides when and for how long each process should be able to perform computations [95]; the algorithms for this range from simple priority-based schemes to proportional-fair algorithms, in which a process can be guaranteed x time units every y time quanta. This can be compared to a virtual economy on each CPU, in which processes receive an allocation of funds, and effectively purchase their CPU cycles [104], or receive ‘shares’ in a machine which can be used to prioritise jobs as on the IBM 370/165 at Cambridge University in 1974 [65]. On multiprocessor systems, these decisions become more complex, because processes can also be migrated between processors for better performance, but at considerable extra cost. Nevertheless, for interactive applications which have soft real-time constraints, proportional-fair schedulers can still be very effective [19]. CPU schedulers also need to avoid using excessive resources themselves when scheduling other processes, particularly when the number of processes is high — as illustrated by the constant overhead scheduler recently integrated into the Linux kernel [76], which shows substantial performance improvements on many multi-threaded benchmark tests.

Distributed applications also suffer from resource constraints, but without the convenience of a central point of control, or even clear knowledge of the resources available. As a result, resources can be difficult to price appropriately. For example, in wireless sensor networks [100] and ubiquitous computing applications, the energy available is limited so resource usage has the double cost of reducing system lifetime and pre-empting other tasks — and some of these tasks such as forwarding data to other nodes are particularly hard to assess from the perspective of a resource-limited, myopic server. The problem of unpriced resources or flat-rate pricing affects not only users but also providers of services, such as Internet Service Providers which then need to overcharge light users to subsidize heavy users [33].

Predicting resource needs in advance is part of the difficulty of job scheduling. Although many metrics have been used to profile computational resource requirements, job schedulers are hard to test because of the difficulty of automatically generating representative workloads [37]. This arises partly because programs do not usually make their resource requests or needs known in advance, and partly because there is little outside support for these requests.

Research into proof-carrying code (PCC) [77] has shown that it is possible to formally prove the resource requirements of program code; consumers of the code can then validate the proof before allowing the code to execute. However, the usefulness of PCC is limited for complex programs whose resource usage is limited by data they process, not their inherent structure. The TINMAN project [74] has recently proposed modifying this approach by instead attempting to predict resource bounds statistically at the source code level before generating resource certificates for validating and monitoring code execution. However, this then imposes extra run-time monitoring overheads which could be avoided with pure PCC.

Control of resource usage is limited partly by a lack of communication between the scheduler and the programs using the resources. For some applications, actions can be planned in advance, especially when the plans involve real, physical resources that are far more expensive or scarce than the cost of the CPU cycles needed to establish an efficient plan. For example, collaborative calendar applications can use automatic conflict reconciliation strategies to save the users from time spent in rescheduling appointments [84], but at the cost of leaving the schedule undecided until the reconciliation is actually performed; similarly, planning frameworks for robotic applications attempt to schedule a plan which meets mission goals, but minimises the cost of power or fuel used [51]. Business applications can also express their needs and obligations as deontic logic contracts [67] so that they can be monitored and enforced. Nevertheless for most computer programs, these plans would be at too high a level and too structured to be directly useful in controlling computational resources.

A few operating system architectures do give applications explicit control over their resource usage, such as XenoServers for Nemesis [85] and the MIT exokernel project [54], but generally only in the sense of hard resource reservation guarantees, and with limited support for explicit resource pricing strategies or integration of external resources.

Reflective middleware architectures attempt to go beyond this, with a more holistic approach to resource management. In these architectures, programmers can explicitly control all aspects of a middleware service. Some architectures such as DynamicTAO [58] allow programs to activate only the essential middleware services they need, enabling them to operate with a very small memory footprint when on resource-constrained devices, by delegating resource control to separate modules. Others such as the Open ORB project [30] provide integrated support for resource control, allowing applications to inspect and reconfigure their resource allocations dynamically.

Some reflective middleware environments offer a simple economic approach to resource provisioning: in Geihs's architecture [41] applications can identify a number of operating levels, each with an associated resource demand tuple (e.g. CPU time and bandwidth) and a utility value showing the value to the user of the application operating at that level. The resource middleware can then attempt to optimise all of the utility functions simultaneously to identify the best mix of applications which meets the resource supply, although in practice it may be more efficient to identify a good but sub-optimal combination instead.

Reflective middlewares can also be used for flexible control of Quality of Service

(QoS) properties of middleware services such as connection multiplexing and redundancy [105]. This adaptive reconfiguration facilitates strategies to limit resource usage, such as automatically unlinking infrequently used application components, albeit at the potential cost of needing to relink them if they were needed again.

Many of these aspects of system control are seen as orthogonal to the conventional procedural program code. Aspect-oriented programming (AOP) provides an alternative mechanism for this control by extending programming languages to allow each aspect to be factorised out and programmed independently, in languages such as AspectJ and Hyper/J [35]. However, Kienzle and Guerraoui argue that some issues such as concurrency and failures interact poorly with AOP, as they need to be exposed directly to the main application for it to make informed decisions in context [56]. Treating resource usage as an aspect could be subject to the same limitations, which are not inherent in resource middleware architectures.

Although resource usage modelling is crucial in many computer systems, it is often impossible for programs to make their resource needs explicit. Even when this is allowed, applications are usually limited to resource reservation schemes, rather than dynamically negotiating their needs — even the most expressive frameworks limit the expression of resource needs to listing a few acceptable permutations. Nevertheless, existing frameworks would have the potential to provide support for the creation of more complex resource control schemes.

2.2 Trust Modelling

Trust management and trust modelling are increasingly important in networked applications, which need to interact with other machines but may be subject to abuse unless they are protected from attack. The term ‘trust’ has many definitions and interpretations, but the focus in this dissertation is on trust as a computational construct, for moderating observations to identify productive actions. In the typology of McKnight and Chervany [71], trust is divided into a hierarchy of *dispositional trust*, *institution-based trust*, *trusting beliefs* and *trusting intention*. The focus of this work is on trusting beliefs about the predictability of other participants’ behaviour, based on observations and leading to implicit trusting beliefs about the integrity of others — the extent to which they behave honestly and keep their promises. In earlier work, Marsh [70] also argued for the use of trust as a formal computational concept.

Trust management has become a generic term for describing security access policies and their associated credentials, in which ‘trust’ becomes a generalisation of privilege and restricts access to trusted actions. Blaze, Feigenbaum and Lacy [10] introduced the term for their PolicyMaker trust management system, which uses a restricted language to link policies, credentials and trust relationships and authorise actions, which they later integrated with existing public-key infrastructures in the KeyNote [9] system. Although this approach to trust management was originally targeted at security systems, discussed in more detail in Section 2.3, it is also generic enough to apply to evidence-based or personal notions of trust.

Trust models can use direct observations to assess the trustworthiness of others, basing trustworthiness on behaviour rather than on information from other sources such as certificates. The resulting trust models are often informal, and based on ad hoc rules for correlating actions with changes in trust. These rules also define the trust categories available, such as ‘Very Trustworthy’, ‘Trustworthy’, ‘Untrustworthy’ and ‘Very Untrustworthy’ [1] or values from 0 to 1 in a Blackjack application scenario which are then partitioned into high, medium and low trust bands [45].

These informal trust models typically have a finite range of trust values from least to most trusted, or possibly a linearly ordered interval. This linearity restricts the expressive power of the models, by restricting their ability to retain a history of past interactions. Although multiple models could be combined in parallel to model different trust aspects of an application, the inherent linearity still remains because of the independence of the separate models. Furthermore, while the mappings from evidence to trust do allow a high degree of configuration, they also make it difficult to predict the system’s overall behaviour or properties without extensive experiments [64].

As the field of trust modelling grows, more formal and more complex models of trust are being developed. These set out to provide formal frameworks for modelling trust, either based on statistical techniques or by constructing generic structures in which to express a range of trust applications.

For example, trust values can formally model the effect of evidence for and against trusting a principal simultaneously, as shown in Jøsang’s [52] Subjective Logic, an extension of the Dempster-Shafer theory of evidence, which is proposed for use in electronic markets by Daskalopulu *et al.* [22]. This model allows uncertainty in a trustworthiness assessment to be considered explicitly, in contrast with simpler trust representations.

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 $(b, d, u) = (0, 0, 1)$. 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; the sum of belief, disbelief and uncertainty 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:

$$\mathrm{E}(\varphi) = b(\varphi) + a(\varphi)u(\varphi) \text{ where } \omega_{\varphi} = (b(\varphi), d(\varphi), u(\varphi), a(\varphi)) \tag{2.1}$$

If it is assumed that past behaviour is a good predictor of future behaviour [93], then this becomes

$$\mathrm{E}(\varphi) = \frac{b(\varphi)}{b(\varphi) + d(\varphi)} \text{ if } b(\varphi) + d(\varphi) \neq 0, \text{ otherwise } \mathrm{E}(\varphi) = k \tag{2.2}$$

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.

The subjective logic also provides natural operators for discounting and consensus: discounting allows principal A to weight observations provided by principal B about C appropriately, by discounting the observations according to A's trust in C; consensus allows two trust triples to be combined together to produce a new trust value, such as when incorporating new observations into existing belief triples.

Although the subjective logic model provides a rich representation, it has a few significant limitations. Firstly, its essential premise is that each principal has a static, binary state: either ‘trustworthy’ or ‘untrustworthy’. Trust triples then represent the belief, disbelief and uncertainty that the state is actually ‘trustworthy’. Thus $\mathrm{E}(\varphi) = 0.4$ means that there is a 40% probability that the principal is trustworthy, based on current evidence, not that the principal is trustworthy 40% of the time. Secondly, a measure of relative atomicity is need when incorporating new observations into existing trust values, to ensure that they are weighted appropriately; this can be difficult to obtain especially when many participants might unknowingly be witnessing the same event, and then combining their observations.

Others propose using explicit Bayesian models to formally derive trust values from observations, especially in Peer-to-Peer networks [21]. This can be achieved by treating each aspect of a client’s abilities such as download speed or file type availability as a Bayesian prior, whose distribution is to be deduced from the observations [106].

While these trust models defined the details of trust calculation formally, other trust models focus instead on generic frameworks for trust modelling, which can be applied to a wide range of disparate applications. For example, the SECURE project [15] (which stands for Secure Environments for Collaboration among Ubiquitous Roaming Entities) attempts to combine all aspects of trust modelling into a single framework, ranging from trust modelling and risk analysis to entity recognition and collaboration models.

The SECURE trust model allows each principal to express their trust policy as a mathematical function which determines their trust in everyone else, in terms of everyone else’s trust assignments. These trust policies can then be combined to produce a consistent trust assignment for all principals; this is expressed as the least fixed point that results when all the policy functions are combined into a single trust function, and is guaranteed to exist provided that the policy functions are all suitably monotone [16]. This model extends the work of Weeks in formalising trust management in security access control systems in terms of least fixed point calculations [107], into evidence-based trust models.

Models and analogies such as the prisoner’s dilemma [60] have often been used to represent the incentive to cheat, and the social effects of this. Experiments suggest that the best strategy in the prisoner’s dilemma is usually ‘Tit for Tat’: cooperating initially, then imitating the opponent’s previous move from then on. However, with imperfect information or measurement error, Generous Tit for Tat which forgives cheating with a probability of $\frac{1}{3}$ is a superior strategy [46].

Studies have also been made in which the trustworthiness of the opponent is known statistically — represented as the ability of the enforcing agency to persuade the opponent to cooperate [12]. (Reputation systems and other mechanisms for encouraging cooperation may be seen as distributed enforcing agencies in this context.) These preliminary studies suggest that underestimating the opponent's trustworthiness tends to harm both participants, though the relative costs of underestimating and overestimating have not been analysed; they probably depend on circumstances. In other words, erring on the side of caution may not be the best strategy. Even the definition of partial trust is open to debate [73]. In some contexts, partial trust implies that the opponent's capabilities, or ability to do damage, are limited (either directly or through the properties of the enforcing agency). In others, it implies that the opponent is trusted to cause only limited damage. This highlights the difference between the absolutist security approach and the statistical economic approach to trust management, though the two are usually combined to some extent.

Systemic fraud must also be prevented, both locally and globally — it should be impossible for an agent to systematically pilfer significant resources, either from another agent [18], or from the society as a whole.

Reputation and Recommendations

Networks of trusting principals often use reputation services and recommendations to allow trust information to propagate between principals [1]. Reputation services combine the observations of a number of principals, to provide a common reference for trust information — analogous to trusted third parties in security systems. Recommendations, on the other hand, are personal observations by one principal about another, which can be passed to a reputation service or directly to other principals.

Clearly, the protection of reputation services from slander is an important consideration too [24], to prevent deceitful participants from maliciously damaging the reputations of others. Conversely, measures are needed to prevent principals with bad reputations from simply creating new identities for future interactions [99].

Trust model and reputation or recommendation services have been used in many applications, such as peer-to-peer file sharing systems [106] — although it has been argued that these cannot be truly decentralized without being vulnerable to Sybil attacks [28] in which a single attacker pretends to have numerous identities. Other applications include agent-based computing environments [53], internet commerce and web service applications [102], and using trust to limit cryptographic overheads where appropriate for Grid computing [5].

Reputation models are also used in other contexts too, such as in data mining web links to compute the reputation of pages with respect to a certain topic in a weblogging community [43]. Unlike the Google PageRank [78] algorithm, this algorithm treats the problem of reputation assessment as separate from information retrieval. Finally, some projects such as XenoTrust [29] seek to combine reputation-based trust with conventional security into a single trust management architecture. Here, a publish/subscribe system is used both for notification of changes in trustworthiness, and for aggregation of reputation information.

2.3 Security and Access Control

Traditional aspects of security modelling are also vitally important for large-scale computer systems to operate safely and reliably. In this dissertation, the most important of these are access control schemes and policy specification, unforgeable certificates and proof of identity.

Access control systems are designed to limit which users can access certain data or resources in a computer system. These range from discretionary and mandatory access control schemes (DAC and MAC) to more recent rôle-based access control (RBAC) models [89]. While DAC and MAC operate by allowing users to grant (or deny) others access to specific objects, and through the checking of security labels respectively, RBAC adds an extra level of indirection: users instead gain access to rôles — usually by presenting other credentials — which then lead on to permission to access data or resources.

This indirection allows RBAC models to support rich, dynamic access control policy, which can be defined and altered independently from the rest of the system. For example, some rôles could be used as prerequisites for activating other rôles, allowing a least privilege principle to be observed. Some RBAC systems, such as OASIS [7] also allow parametrized rôles, delegation in which some users can grant extra rôles to others, and dynamic revocation of privileges if their preconditions fail.

Policies can also be used to control access to resources at the lower levels of a system, such as to implement QoS guarantees in the management of Differentiated Service (DiffServ) networks [68]. The policy specifications are then compiled from their textual representation into low-level commands for each routing device.

Conventional trust management systems assume that trustworthiness is known correctly at the time when it is used. The degree of trust is then represented in various ways: as a number in economic models [12], as membership of a trust class in privacy systems such as PGP [10] and in other trust frameworks [1], or as membership of a rôle in access control systems [7]. 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 either, since these conditions are very difficult to express as security policies.

Access control systems often depend on signed credential certificates to support distributed operation efficiently. These certificates are electronically signed by the issuer, and used as tokens of authorisation or rôle membership. Because the certificates are prohibitively difficult to forge, they can often be used even when the issuer is uncontactable, e.g. because of network failures, to authorise a user — as long as the issuer is known to be trusted in this context.

Electronic societies need to protect against malicious agents causing damage or stealing resources. This is traditionally achieved by restricting the abilities of agents, as in the Java sandbox model [44]. Here, program code is digitally signed by the author, and a local access policy file specifies which authors' programs can access which resources outside of the sandbox. Unsigned programs or programs from unknown sources are then given no outside access by default. A more difficult task is protecting agents from each other while still allowing them to interact usefully; this requires a combination of trust management tools, and cautious design.

Trust management has also been approached from a number of formal perspectives. Palsberg and Ørbæk have demonstrated how trust annotations can be applied to higher-order languages such as the λ-calculus [79], in such a way that they can be analysed statically before the program is actually evaluated. Other formal systems, such as the boxed-π calculus [92] and ambient calculi [17] can also be used to wrap untrusted code and enforce security policies.

combien
calculi.

Certificates and Signing

The use of certificates depends strongly on effective schemes for digital signatures and proof of identity. In public-key cryptography, each principal has two keys: a public key which is freely distributed, and a secret, private key that only that principal knows [72]. Using the public key, principals can encrypt messages that can only be read by the holder of the private key. Conversely, the private key can be used to digitally sign messages, with a signature that can be verified by any holder of the public key. Although the private key could in principle be discovered through exhaustive testing, a sufficiently long key is computationally infeasible to crack. On the other hand, cryptographic security in messages does not offer a perfect guarantee against key compromise, as the key could conceivably be obtained by other means such as installing a rogue program on the signer's computer to extract the unencrypted key from main memory. Institutions such as banks try to avoid this limitation by providing cryptoprocessors for customers, in which the secret key is physically protected inside a special processor which can be used only for cryptographic tasks. However, many of these processors still have vulnerable interfaces which can be used to extract information about their private key and thus crack it [11]. Even with effective cryptoprocessors, flawed processes may be persuaded to sign messages without the user's knowledge, unless a specially secured terminal is used [8], so a digital signature provides high but not perfect confidence in the validity of a message.

The X.509 Public Key Infrastructure (PKI) [48] specifies a standardised format for digital certificates, primarily for use in identifying principals (as discussed in the following section), but which has also been used for other purposes such as expressing rôle membership certificates in RBAC schemes. These certificates can have many attributes for carrying information, and are signed by the issuers to prove their authenticity.

Signing is useful not only for certificates and messages, but also for providing indirect guarantees of authenticity. For example, digital signatures can be used to generate post-unforgeable transaction logs to prove that a computation was actually performed at a certain time, but without communicating all of the intermediate results. This is achieved by generating a secure hash[1] of the intermediate results, and sending only the signature of this to a well-known trusted

[1] A secure hash function such as SHA-1 [32] condenses a message into a small, fixed-length message digest. The digest is secure in the sense that it is computationally infeasible to discover a message which generates a given digest, or two messages that produce the same digest. Thus the digest can act as a fingerprint for the original message.

third party to hold as proof. Successive hashes can be chained together, by incorporating the previous hash into the new signature, making this a secure audit mechanism that can be used to prove that computations were indeed performed.

There are also other techniques for proving that computations have been performed, such as multi-party secure computation, which distributes a computation over many machines, but prevents any minority from altering the result. In principle, multi-party secure computation would allow secure auditing to take place, but this could be prohibitively expensive to implement [47]. Furthermore, this would still not protect against cheats falsifying the original inputs, as long as this cheating was achieved in real time.

A particularly important shared computation is fair exchange of information between two computers, which seeks to ensure that the exchange is either completely successful or else aborted so that neither party holds the other's information [69]. For example, this can be used for electronic payment schemes, to ensure a process in which all payments are acknowledged with signed receipts. Section 3.3 covers fair exchange protocols in more detail.

Proof of Identity

Proof of identity is also needed for security systems, to ensure that principals can recognise and identify those they interact with. On the other hand, this drive for identification clashes with some principals' need to remain anonymous. To some extent, both of these problems can be solved using public-key cryptography, by using principals' public keys to identify them. A few well-known principals can then be established as Certificate Authorities (CAs) which sign others' public keys, issuing X.509 certificates that link them to their real-world identities. Each CA acts as a Trusted Third Party (TTP) in the sense that its users all trust it to generate certificates honestly and unambiguously, and to keep its private key safe from being compromised. These CAs are conventionally linked hierarchically, with one root CA authenticating other CAs which can in turn authenticate others [34]. This hierarchy can be extended further by users themselves, who can use their primary identity to authenticate their other identities. Then if one of the user's secondary private keys were compromised, the owner could publish a revocation certificate and create a new identity to replace it. In the short term, access to the compromised keys might allow damage to be done, but ultimately it would not reflect a total loss of the principal's identity.

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' [10] 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.

On the other hand, principals that need to remain anonymous can generate fresh public/private key pairs to identity themselves pseudonymously to others, unlinked to their other identities. Of course, these nonce-principals would be unknown to others and thus untrusted; in security systems, untrusted principals would usually have the lowest trust possible, to prevent others with minimum trust from generating fresh identities instead to gain trust and access to resources. Nevertheless, they could establish some trust from others by paying money to them anonymously using untraceable electronic cash [20] or other transferable securities similar to the code number of a mobile phone credit top-up card. Alternatively, they could use environmental proofs of identity to establish trust initially with only partial loss of anonymity — such as knowledge of a guest username and password, possession of a delegated rôle membership certificate, or location-based identification such as an IP address on a secure intranet — although there is ongoing research into principal identification by correlating behaviour and other evidence [91].

There is no universal cure for untrustworthy agents; however with long-lived principal identities (whether linked to real world identities or not), trustworthiness can be characterised, to provide an incentive not to cheat. Through the use of reputation systems, virtual social institutions [25], or other enforcement agencies, this can be achieved. However, this necessarily leads to reduced anonymity for participants, and makes it more difficult for newcomers to enter relationships (since they might be incorrigible cheats, masquerading as newcomers) [99]. Nevertheless, there may be even greater rewards for this sacrifice of anonymity than previously believed [23], because large networks of trust can then be generated easily between previously unfamiliar participants, as characterised by Metcalfe's Law, which states that the usefulness, or utility, of a network varies with the square of the number of users.

2.4 Self-organising Distributed Systems

This section reviews techniques and tools for organising computations in large-scale distributed systems, ranging from centrally controlled computer clusters, through loosely coupled scientific 'Grid' applications, to peer-to-peer techniques for ad hoc collaboration between strangers. It focuses on how interactions can be controlled and monitored in these systems, through explicit contractual agreements and economic modelling.

Computational economies offer a relatively new, but promising approach to distributed cooperation. In the Spawn system [103], each task has a budget, and uses this to bid for an idle CPU in a network of workstations. Important tasks are assigned a larger budget than other tasks, which they can use for priority access to resources. By choosing an appropriate bidding strategy, tasks can optimise their use of funds. This has led to the stride scheduling proportional-share algorithm for CPU process scheduling, based on a ticket system.

More recently, interest in shared application servers for large corporations has led to plans by companies such as HP Laboratories (in the eOS project [110]) to support automatically migrating commercial applications on a global scale, based on resource availability. Similarly, Intel Research is sponsoring the PlanetLab project 'for developing, deploying, and accessing planetary-scale services' [83], while other scientific and commercial consortia such as the Globus alliance [38] are developing generic toolkits for engineering Grid applications. The EU and CERN are also developing a DataGrid project [13], to act as a huge computing resource, for scientific and commercial applications, while the GRACE grid project [14] uses a distributed computational economy to prioritise its tasks.

Underlying most of these plans is the idea of a huge, global computing network, which will become a basic resource, comparable to the electricity grid. Computers on the Grid would all be reasonably trustworthy, paid for by application service providers, who would in turn charge for the facilities they offered, probably using standardised commodity pricing schemes, again comparable to electricity markets. Nevertheless, current Grid applications are mainly repetitive parameter-sweep tasks operating over trusting networks.

The main challenge is not simply allowing distribution of code — systems such as PVM already allow this ([42], outlined below) — but discovering how to build programs which can take advantage of this automatically [66]. Ideally, independent services should discover each other dynamically, and be able to subcontract tasks wherever appropriate. This necessarily introduces the issue of trust *between services* (rather than between services and the underlying network), as well as the need for a language in which agents can make their needs known.

First steps towards this have been taken with agent languages such as KQML, the Knowledge Query and Manipulation Language — a language and protocol for exchanging information and knowledge, part of the larger ARPA Knowledge Sharing Effort [61]. Some have also suggested making these agents market-aware [108], but again there is a lack of suitable tools for developing them. Indeed, the strengths and weaknesses of these multi-agent systems are seldom critically analysed [66] — a few studies have been performed for distributed engineering control applications [111] to help rectify this, but only for simple problems.

The Mojo Nation project was the first real example of a public computational economy (until February 2002), though the services initially available were limited to file sharing and distribution [109]. In this system, users pay to locate and retrieve files using the 'Mojo' currency, and are paid in turn when they supply files to others. According to the authors, accounting and load balancing are also decentralized, but it seems that the global name service is centralized in one or two 'metatrackers'. (In October 2000, the Mojo Nation network failed and had to be modified, because more than 10 000 new users tried to join the network on a single day.) It also seems that only content-related resource usage is accounted, exposing the higher level services such as accounting to denial-of-service attacks. The successes and failures of this project illustrate the power and complexity of large-scale computational economies, and also the need for realistic simulation to assess the scalability of a system.

Other peer-to-peer file sharing services have also tried to use market mechanisms — either formally or informally – to ensure that participants provide resources as well as consume them. For example, the Free Haven anonymous storage service [26] uses an economy in which participants are effectively paid for providing shares (fragments of files) to others, and charged for losing blocks. Projects such as SETI@home [59] have made use of large-scale distributed networks to solve computational problems, but these are currently limited to computational tasks that are based on distributing work units for (essentially) off-line processing and eventual hierarchical submission of results. However, these projects support only a very limited range of economic and computational interactions; general purpose tasks would need richer behaviour.

Contract Formation and Representation

A contract is an agreement between two or more parties about the actions they are to perform; similar contracts also regulate behaviour when computers cooperate. These contracts range in sophistication from simple, hard-coded communication protocols, to the exchange of entire programs to be executed.

The simplest computerised contracts are communication protocols, which are rules which allow computers to unambiguously interpret each other's messages. These protocols allow only simple, circumscribed actions to be performed, such as making a copy of a particular file, or authenticating a user's identity over a network. In areas such as access control [7], this enforced simplicity enhances security by allowing only predefined actions to take place. However, for tasks such as flexible distributed computation, a contracting system would need to allow for the dissemination and use of new software on the fly.

For example, PVM (Parallel Virtual Machine) is a software system which links separate host machines into a single virtual computer, for use in scientific simulations and other applications [42]. In PVM, it is the user's responsibility to initially distribute the program code to each node, before executing it in parallel through PVM. The control program (also written by the user) is then responsible for distributing the active tasks between nodes, and collating the results. In this scenario, nodes have two conceptual levels of contractual obligation:

1. to execute subroutines as directed by PVM, at the implementation level,
2. to faithfully contribute results towards the overall computation, at the interface level; this is not made explicit, but is important to the user.

While the PVM protocols specify the mechanics of distribution, it is the control program's responsibility to organise what might be called the social interactions of the nodes so that they cooperate effectively. The contractual agreements underlying this are PVM's hard-coded protocols, and the inscrutable executable subroutines, respectively. PVM implicitly assumes that the user is trustworthy and the program is correct, thus a malicious or faulty PVM module can damage the network. Furthermore, accounting of resource usage is difficult and not explicitly supported by PVM, hindering efficient use of resources.

This suggests that an intermediate level of contract might be desirable — less rigid than the hard-coded protocols, but more comprehensible than ordinary executable code, and subject to introspection.

Making Contracts Explicit

This section reviews how contracts have been made explicit in existing systems, and highlights the distinction between the contract itself and the actions involved in performing it.

The archetypical example of this is the Contract Net protocol [96], in which computer nodes advertise tasks which need to be done, or bid to perform them. The initial negotiations involve only high-level descriptions of the tasks, and nodes are usually selected according to their speed and suitability for the particular task. Worker nodes may also subcontract their calculations, if necessary.

The protocol was at first designed for remote sensing applications, but has been used in many other domains too. The original implementation assumed that all participants were trustworthy, and all tasks important; more recently, Sandholm and Lesser [88] have explored levelled commitment and breakable contracts, though still only with trustworthy agents, and Dellarocas and Klein [25] have postulated the need for electronic social institutions to provide robustness in an open contract net marketplace.

Others have analysed the relevance of business contract theory in this domain. On the one hand, computers can be used to partially automate business contracts, through enforceable online transaction systems [49] and automated contract negotiation [57]. On the other hand, traditional business accounting controls, such as double entry bookkeeping, segregation of duties and auditing techniques, can be applied to computerised contracts [99], instead of reinventing them from nothing. Similarly, business theory suggests that both computational and mental 'transactional costs' — the overheads of performing contracts — should be included in a comprehensive contract model, as does human-computer interaction research into unintrusive or distraction-free computing [40].

Explicit contracts are also used for specifying network service needs, in the form of Service Level Agreements (SLAs). Although these are traditionally limited to paper-based contracts, other projects such as TAPAS [75] are investigating electronically enforced SLAs.

This leads to the problem of assessing whether a contract has been correctly performed. The primary difficulty here is the lack or cost of this information — which is often why a contract was formed in the first instance. For some tasks, such as NP complete problems [80], generating an answer is far more difficult than checking its correctness. However, this would not help verify original information from suspect sources, unless corroborating information could be found elsewhere. Studies in linguistics have investigated the complexity of lies, and the difficulties of defining them [62, pp. 71–74]. There is also the larger problem of differentiating between intentional and unintentional breaches of a contract, and situations where the contract was improperly constituted in the first place [4].

Many of these problems can never be completely resolved, or would be too expensive to rectify. Nevertheless, high-level contracts can help to formalise relationships between computers, improving accountability and providing a level at which trust analysis may be feasible.

2.5 Summary and Nomenclature

For computer systems to manage their resources appropriately, they need to explicitly codify their resource usage. This applies equally to local resource management and to resource management in distributed systems, which can be seen as cooperating networks of autonomous nodes. To express this effectively, each node also needs to model the cost and benefits or priority of its contracted tasks, as well as the trust it has that other nodes will cooperate with it. While existing systems do support contracted operations, they are very limited in the range of interactions possible, and in their abilities to express the link between a task's resource needs and its priority. Thus there is a need for a consistent framework to control and monitor resource usage, for trust-based interactions in distributed computing environments.

Throughout the rest of this dissertation, following concepts are crucial: A computational *contract* defines an exchange of resources between a client and a server; the rate of exchange is defined by its *accounting function*. *Resources* include computational resources such as CPU time and network bandwidth, and also payments and external constraints. *Trust* represents the expected behaviour of a participant in performing contracts, in economic terms; the *trust model* is the operational framework for identifying worthwhile contracts. Contracts are useful in both competitive applications and as *service contracts* which use their accounting functions to support cooperative distributed services.

The following chapter (Chapter 3) defines a general purpose contract framework, for modelling interactions in both centralized and distributed systems. Chapter 4 then integrates trust modelling into the framework and proves the model's resilience to attack, before illustrating its usefulness in a compute server application. Later chapters show that the framework's techniques can also be applied to other tasks such as automating human-computer interactions and controlling distributed service provision.

Chapter 3

A Framework for Contracts

A contract framework lets computer systems formally agree to the tasks they undertake, taking into account the resources they need. Payments or resource exchanges can also be expressed in the same resource model, so that contract compliance can be defined and measured. Contracts need to be negotiated and signed electronically too, to ensure that both parties accept their terms as binding. Finally, a trust model will be needed to help identify good contracts from reliable participants automatically.

The contractual approach is useful for a wide range of applications, such as task management on a single computer, or distributed computation in ad hoc virtual communities where contracts protect against service disruption by unscrupulous participants. This chapter establishes the framework for contracts and introduces its novel and expressive resource model, while Chapter 4 goes on to show how trustworthiness and reliability can be modelled as contract transformations, both within secure local domains and in virtual communities connected only by personal recommendations. Thus the contract model contributes an intermediate representation for tasks that acts a bridge between inscrutable computations and measurable utility.

Instead of insisting on a substantial trusted third party infrastructure for the contracting framework, we allow subjective contract assessment, and propose the use of a rich trust model to allow these subjective assessments to be combined. Although this allows clients and servers to try to lie about contract compliance, the opportunities for this are very limited (since actions such as contract payment are cryptographically signed by both parties, acting as contract checkpoints: see Section 3.3.1) and the benefits are bounded by the trust model's safety properties (see Section 4.2.3).

3.1 Motivation

Computer systems need to control access to their resources. These resources may be scarce or expensive, and so urgent or important tasks should take precedence in using them. However, even the most important tasks need to be monitored to stop them consuming more resources than intended. This monitoring should be automatic, to allow computer systems to regulate themselves with little external intervention.

Traditionally, access control permissions and task priority levels are used to decide which tasks are run, and when. These decisions are implemented at many layers, ranging from interrupt request prioritization at the hardware level, through the operating system's task scheduler and file permission controls, to middleware for high level access control of services.

All of these mechanisms trade off flexibility of control against the overheads they impose; more powerful control mechanisms are also more difficult to analyse to ensure code safety and liveness, which guarantee that the system behaves correctly and yet makes progress. However the resulting power allows for fine-grained control, while simplifying resource administration — for example in rôle based access control systems, policy authors can associate a privilege with a group of users acting in a certain context, rather than simply with local user identities. Furthermore, this administration and authentication can be performed remotely in a different domain, then applied locally.

For tasks where security is paramount, a permission based approach to control is appropriate. On the other hand, when resources are scare, this is not enough: two tasks might be permitted to access the same resource, when only one can actually do so. In a sense therefore, access control systems are designed for restricting access, not enabling it. What is needed is a mechanism for authorizing access based on resource consumption.

For authorized resource consumption, the client (seeking to use the resource) must negotiate with the server (offering the resource) to match its needs to the resources available. The server's authorization can then be expressed as an explicit contract to supply resources. Conversely, the client should also enter into the contract, promising to abide by the resource allocation, and perhaps to provide payment or other resources in exchange for those of the server.

Expressing services through contracts would allow computer systems to plan for their resource usage and consumption, and allow scarce resources to be apportioned between claimants appropriately. Although clients cannot always precisely predict their resource needs in advance, contracts would still guarantee a minimum quality of service, and could also express the terms of exchange for extra resources used. By assigning a value to each resource type — either statically or using a market value — servers could also differentiate between contracts in assigning extra resources beyond the minimum level promised.

The applications of this contract framework would include not only selling CPU time on compute servers and similar services, but also extending traditional notions of access control to take into account resource *consumption* too. For example, a resource-constrained sandbox could be created for running untrusted code, and interruptions of the user's time by programs in the background could be limited by treating this as a scarce, contractable resource.

Clearly, contract specifications would need to be constrained, to prevent them executing arbitrarily code which would make contract analysis prohibitively difficult. Furthermore, untrustworthy or unreliable clients and servers might break their promises, in breach of contract. As a result a trust model would be needed, for monitoring the contract framework, and to help in deciding which contracts should be accepted.

3.1.1 Applications

There are two broad classes of applications of the contract framework: service oriented and user oriented applications. Distributed services need to perform computational tasks remotely for their clients across a network, possibly because they have special resources (such as extremely fast or underused processors) or a special location in the network, or because the same resources can be reused for many clients. In contrast, user oriented applications must focus on controlling many tasks vying for the user's scarce time and attention.

To test the contract framework motivated above, three applications were developed:

1. a compute server which runs others' programs for money in a market economy,

2. a collaborative PDA application, in which the user's time is protected by resource contracts, and

3. a distributed service to detect composite event patterns in a publish/ subscribe network, which migrates pattern detectors through the network towards event sources.

These applications exercise the contract framework, and test its effectiveness as a general purpose resource control framework, able to consider the suitability of complex contracts and take into account trust dynamics in many application domains. Although each application is independent and emphasises a different aspect of the framework, they all use the same underlying infrastructure, together exercising all facets.

3.1.2 Contract Executability

If contracts are to control the resource usage in computer systems, then the resource overheads of the contracts need to be taken into account too. This can only work effectively if the contract framework can limit or predict the resources used in assessing a contract, by controlling the granularity of analysis and the level of expressiveness of contracts.[1]

On the one hand, contracts need to be expressive enough to represent concepts such as tiered resource exchange rates and other stateful provisions, resource prices varying with market conditions, and contract prerequisites such as resource deposits and the corresponding returns on successful completion.

On the other hand, contract terms which are too expressive or even Turing complete may themselves consume resources unpredictably, defeating their purpose of controlling consumption; in an extreme case, maliciously designed contract terms could be used to overwhelm a server in a denial of service attack. Furthermore, if the contract framework is to be able to analyse contracts before agreeing to them, it must be possible to simulate the contract terms in isolation without performing the contract action.

The aim is therefore to choose a contract representation with predictable resource bounds, yet which is expressive enough to allow rich, stateful provisions. Furthermore, very little (if any) interaction should be allowed between the contract itself and the action performed under it, in order to allow contracts to be simulated in a planning framework.

[1] The alternative would be a global certification framework, in which programs were certified only if they controlled their own resource usage. However, even here, a methodology would be needed for proving or enforcing the effectiveness of this control.

3.1.3 The Need for Trust

Even if contract terms are constrained in complexity, they might not always be honoured by all parties to the contract. A trust model (Chapter 4) can be used to compensate for this: by storing up information about each participant's previous behaviour, it can predict whether it would be worthwhile to enter into a new contract, or reject it as probably unprofitable. This information could include not only direct observations of contract compliance, but also indirect sources of information such as trust recommendations based on others' observations. The same trust model could also be used to decide whether to continue supporting existing contracts when the other party was violating the contract terms.

Contract violations might be malicious or accidental, caused perhaps by a failure or a lack of resources. From the perspective of the aggrieved party, these have the same effect, and could thus be treated as equivalent in a subjective trust model. For reliability, the trust model should support disconnected operation, although recommendations from other participants or well known trusted agencies may be used for bootstrapping the trust model with new participants.

Ultimately, the trust model should aim to preserve the safety of the resource system, by preventing attacks which consistently milk the system of resources — such as engaging in many small, successful contracts in order to win even more resources back by cheating on a large contract — while ensuring liveness by facilitating honest behaviour and bootstrapping trust values. Thus the trust model should oversee contract negotiation and execution, restricting participants' access to resources based on their reliability.

3.2 Contract Specification

A contracting framework allows computerised resources to be controlled consistently and rationally. This section defines these contracts, how they express their resource needs, and how they are negotiated and managed. When contracts are interpreted, their conditions are moderated by input from the trust model defined in Chapter 4, to discourage cheating.

A contract defines an exchange of resources between a client and a server. Thus a contract specifies the following:

The server promises to offer resources through the contract.

The client will be responsible for paying the server for resources used under the contract.

Resource requirements specify the minimum levels of resources the server will offer.

Accounting function defines the terms of exchange between server and client resources.

Contract action will use the resources which the server offers. This would usually refer to program code provided by the client (such as a Java method signature) which should be executed under the contract.

The structure of resources is defined hierarchically, to allow resources to be described in summary at a high level, and also in full detail.

Resources $\mathcal{R}$ is the space of all resource types. In a typical system, these might include computational resources such as CPU time, network bandwidth and storage usage. Money would also be treated as a resource of exchange, allowing financial contracts.

The space of resources is subdivided by resource type and subtype. For example $\mathcal{R}_{\text{CPU}} \subseteq \mathcal{R}$ denotes CPU resources, while $\mathcal{R}_{\text{CPU},\mathtt{ix86}} \subseteq \mathcal{R}_{\text{CPU}}$ denotes CPU resources on Intel processors. These subspaces allow contracts to specify the resources they require as specifically as necessary, while allowing the contracting server flexibility when appropriate.

Cumulative resource usage $\mathcal{R}_U$ shows cumulative usage of resources of all types over time. Each resource usage entry $u \in \mathcal{R}_U$ is a function of time and resource type, which returns the total resources used until the time of interest. For example, $u(t_1, \mathcal{R}_{\text{CPU}}) = 10.5$ means that 10.5 normalised units of CPU time were used until time t_1, with $u : \mathbb{R} \times \mathbb{P}(\mathcal{R}) \rightarrow \mathbb{R}$.

A cumulative resource usage function attributes resources to all matching categories. Thus $u(t_1, \mathcal{R}_{\text{CPU}})$ would include all CPU resources — both those for specific architectures such as $\mathcal{R}_{\text{CPU},\mathtt{ix86}}$ and those for no specific architecture. However, $u(t_1, \mathcal{R}_{\text{CPU},\mathtt{ix86}})$ would comprise only resources of type $(\text{CPU}, \mathtt{ix86})$.

Having formalised the space of resources and their usage over time, we can re-express a contract's resource requirements and accounting function formally: the resource requirements of a contract c are given by $\mathsf{require}(c, t) \in \mathbb{P}(\mathcal{R}_U)$, the collection of resource combinations that would satisfy the contract. So a given resource pattern $u \in \mathcal{R}_U$ satisfies the contract's resource requirements until time $t \in \mathbb{R}$ if $u \in \mathsf{require}(c,t)$. The resources actually used are then passed to the contract's accounting function $\mathsf{account}(c) : \mathcal{R}_U \rightarrow \mathcal{R}_U$ to determine the minimum level of resources that should be provided by the client in return. Coupled with the resource hierarchy, this allows resource requirements to be made as general or specific as necessary.

When a contract is being executed at time t, $\mathsf{used}(c,t) \in \mathcal{R}_U$ represents the resources used by the client, and $\mathsf{paid}(c,t) \in \mathcal{R}_U$ shows the resources repaid. Lastly, $\mathsf{offered}(c,t) \in \mathcal{R}_U$ are the resources that the server had made available to the client until time t.

A client need not use all the resources made available to it under a contract, thus $\mathsf{used}(c,t) \leq \mathsf{offered}(c,t), \forall t \in \mathbb{R}$, where the ordering on $\mathcal{R}_U$ is the natural, pointwise ordering. Addition and subtraction are similarly defined pointwise. However, the resources offered must comply with the contract requirements.

Contract compliance measures the extent to which a server or client is complying with the terms of the contract. This is not simply a boolean yes/ no answer, but instead expresses the resource shortfall or surplus, compared to the contract terms. If more resources are offered in a compliant contract, it remains compliant:

$$\forall u, v \in \mathcal{R}_U,\ u < v \ \wedge\ u \in \mathsf{require}(c,t) \Rightarrow v \in \mathsf{require}(c,t)$$

A server complies with contract c at time t if $\mathsf{offered}(c,t) \in \mathsf{require}(c,t)$. The server's contract shortfall or surplus is s if

$$\begin{gathered} s + \mathsf{offered}(c,t) \in \mathsf{require}(c,t) \text{ and} \\ \forall u < s,\ u + \mathsf{offered}(c,t) \notin \mathsf{require}(c,t) \end{gathered}$$

If $\mathsf{paid}(c,t)$ represents the cumulative resources provided by the client in a contract until time t, then client compliance is measured as

$$\mathsf{paid}(c,t) - \mathsf{account}(c)(\mathsf{used}(c,t))$$

If contract terms are not met, the other party is not obliged to continue with the contract although it may be in its interests to do so, in case the non-compliance is accidental, or if it has already allowed for a degree of non-compliance in analysing and accepting the contract. (This aspect of trust is defined in more detail in Chapter 4.)

Assessment of contract compliance, and attribution of responsibility for non-compliance, is often local and subjective. For example, if a client is obliged to make a financial payment to a server, the server might simulate a communication failure and blame the client for non-compliance, while the client would blame the server. No third party could distinguish which version of events was accurate and determine whether to distrust client or server, without extra information.

In the simple example above, communications could be monitored by a trusted intermediary to determine responsibility. However, other disputes would not be as easily resolved: a contract action typically refers to code to be executed under the contract. If the client and server disagree on whether this code has been executed correctly, external validation might be impossible or at least prohibitively expensive: the server would have to have recorded all relevant communication, and reliably logged this information with a trusted third party, such as by sending it a secure hash of the data [32] which acts as an unforgeable data fingerprint and makes it impossible to falsify retrospectively. To validate its claim, the third party would then need to simulate the operation of the entire contract again, at considerable expense.

3.2.1 Contract Specification in a Simple Compute Server

As a concrete demonstration of contracts, resources and compliance, we consider the example of a simple compute server, which offers to run programs for others in exchange for money.

For simplicity of exposition, we assume that the space of resource types $\mathcal{R}$ has just three elements; $\mathcal{R} = \{\text{CPU}, \text{I/O}, \text{money}\}$ and is partitioned into three independent subtypes: CPU resources $\mathcal{R}_{\text{CPU}}$, input/output bandwidth $\mathcal{R}_{\text{I/O}}$ and money $\mathcal{R}_{\text{money}}$. In a real system, these subtypes would probably be further subdivided, e.g. by location and currency.

In this system, a cumulative resource usage entry u can chart how each resource type is used over time. For example, if 1 CPU unit and 2 units of bandwidth were used per second for the first 10 seconds, and nothing else, this would correspond to the usage graph a shown in Figure 3.1.

A contract c_1 might give its resource requirements $\mathsf{require}(c_1, t)$ as a number of checkpoints, such as 'at least 1 CPU unit per second for the first 5 seconds, plus an extra 10 units within the first 10 seconds'. The CPU resources needed to satisfy this contract for the first 5 and first 20 seconds are shown in Figure 3.2.

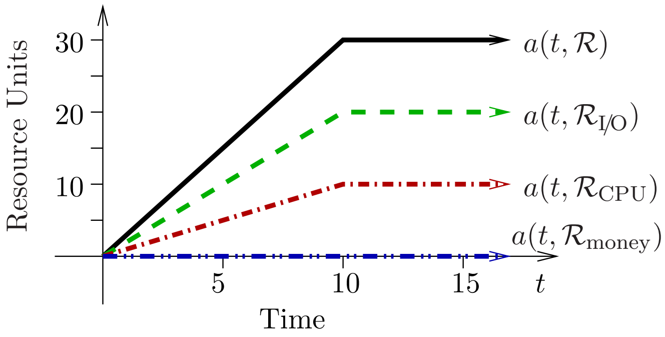
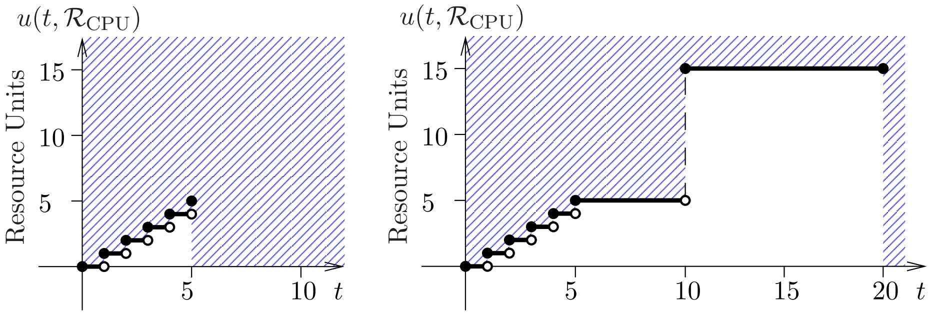


Figure 3.1. Graph of resource usage $a \in \mathcal{R}_U$ over time



(a) After 5 seconds: require$(c_1, 5)$

(b) After 20 seconds: require$(c_1, 20)$

Figure 3.2. Possible values of $u(t, \mathcal{R}_{\text{CPU}})$, if u satisfies contract c_1

The first graph shows that at the end of each second, an extra unit of CPU resources is required to satisfy the contract. The shaded area of the graph shows the range of values associated with compliant resource offerings offered$(c_1, 5)$ satisfying require$(c_1, 5)$. For example, the graph a of Figure 3.1 clearly satisfies contract c_1 for the first 5 seconds, and lies wholly within the shaded area of Figure 3.2(a). However, it no longer satisfies the contract after 20 seconds, since $a(10, \mathcal{R}_{\text{CPU}}) = 10$, but c_1 requires at least 15 CPU units within the first 10 seconds. This can be seen graphically too, since $a(t, \mathcal{R}_{\text{CPU}})$ lies beneath the shaded area of Figure 3.2(b) for $t \in [10, 20]$.

The graphical representation conveniently summarizes the requirements of a contract, but risks oversimplifying matters: no usage graph which extends outside the shaded requirements will be compliant, but the reverse is not always true. For example, supplying 15 CPU units within the first second and nothing else will not satisfy c_1, despite lying inside the shaded requirements area. This is because requirements definitions are not simply pointwise, but can depend on the entire usage graph. Thus $\mathsf{require}(c_1, t)$ should be defined formally as:

$$
\begin{array}{rrl}
u \in \mathsf{require}(c_1, t) \iff & u(1, \mathcal{R}_{\mathrm{CPU}}) - u(0, \mathcal{R}_{\mathrm{CPU}}) \geq 1 & \text{if } t \geq 1 \\
& \text{and } u(2, \mathcal{R}_{\mathrm{CPU}}) - u(1, \mathcal{R}_{\mathrm{CPU}}) \geq 1 & \text{if } t \geq 2 \\
& \text{and } u(3, \mathcal{R}_{\mathrm{CPU}}) - u(2, \mathcal{R}_{\mathrm{CPU}}) \geq 1 & \text{if } t \geq 3 \\
& \text{and } u(4, \mathcal{R}_{\mathrm{CPU}}) - u(3, \mathcal{R}_{\mathrm{CPU}}) \geq 1 & \text{if } t \geq 4 \\
& \text{and } u(5, \mathcal{R}_{\mathrm{CPU}}) - u(4, \mathcal{R}_{\mathrm{CPU}}) \geq 1 & \text{if } t \geq 5 \\
& \text{and } u(10, \mathcal{R}_{\mathrm{CPU}}) - u(0, \mathcal{R}_{\mathrm{CPU}}) \geq 15 & \text{if } t \geq 10
\end{array}
$$

This is obtained by combining 6 independent conditions, one for each line of the definition:

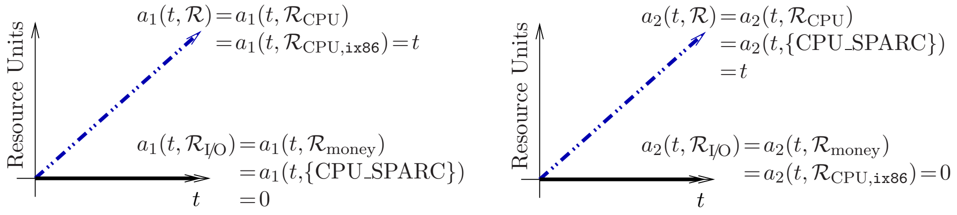
$$
\begin{array}{rcl}
u \in \mathsf{require}_1(c_1, t) & \iff & u(1, \mathcal{R}_{\mathrm{CPU}}) - u(0, \mathcal{R}_{\mathrm{CPU}}) \geq 1 \text{ if } t \geq 1 \\
u \in \mathsf{require}_2(c_1, t) & \iff & u(2, \mathcal{R}_{\mathrm{CPU}}) - u(1, \mathcal{R}_{\mathrm{CPU}}) \geq 1 \text{ if } t \geq 1 \\
& \vdots & \\
u \in \mathsf{require}_6(c_1, t) & \iff & u(10, \mathcal{R}_{\mathrm{CPU}}) - u(0, \mathcal{R}_{\mathrm{CPU}}) \geq 10 \text{ if } t \geq 10
\end{array}
$$

Then, assuming CPU resource usage never decreases,

$$
\mathsf{require}(c_1, t) = \sum_{i=1}^{6} \mathsf{require}_i(c_1, t) \quad \text{with addition defined naturally.}
$$

Both the graph and the formal definition show that $\mathsf{require}(c_1, t_f)$ places no constraints on its elements $u(t, \mathcal{R}_{\mathrm{CPU}})$ for $t > t_f$. In other words, the requirements function constrains only past, not future usage at each moment. This can be seen in Figure 3.2(a) where all usage values are shaded for $t > 5$.

Why the elaborate system with two time axes? Firstly, this allows for the simplest representation of contract requirements, since the time of assessment t_f and the assessment history timeline t are explicitly differentiated. Secondly, the ability to refer to the past allows compliance to depend on both past behaviour and current resource provisioning. Finally, independent time axes allow contract requirements to change over time. This allows maximum flexibility in contract specification, and allows a contract's terms to be changed on the fly without changing the compliance history. In our example, if the server does not provide enough resources for the client, it violates the contract and continues to do so indefinitely. However, the client and server could negotiate a penalty fee, and change the contract terms to allow the penalty to offset the original shortfall. This would allow the contract to become compliant again, but should



(a) a_1 offers an `ix86` CPU

(b) a_2 offers a SPARC CPU

Figure 3.3. Illustration of Resource Generalisation

still not alter the fact that it was violated earlier — otherwise past observations and testaments about the contract's progress would have to be revoked. Instead, by changing only the contract's future conditions, past violation could be reconciled with present compliance.

The contract's provisions for resource shortfall need not have been added retrospectively; alternatively they could have been included explicitly in the original contract terms. Thus a resource shortfall would have led to only a period of non-compliance, until suitable reparation was made and compliance was reestablished.

The example given above is necessarily simplistic and incomplete; it does not address aspects of the contract framework such as resource generalisation and client compliance, nor practical considerations such as participant trustworthiness and contract selection. Many of these are covered in a more comprehensive computer server application in Section 4.3, but the contractual aspects are also illustrated below.

An important feature of the contract formalism not addressed in our simple example is resource generalisation. While this might not be sensible for combining heterogeneous resource types such as CPU time and bandwidth, it can be extremely useful for aggregating different resource subtypes. In contracts, this allows resource requirements to be expressed as generally or as specifically as needed. For example, if $\mathcal{R} = \{\text{CPU\_}\texttt{ix86}, \text{CPU\_SPARC}, \text{I/O}, \text{money}\}$ with $\mathcal{R}_{\text{CPU},\texttt{ix86}} = \{\text{CPU\_}\texttt{ix86}\}$ and $\mathcal{R}_{\text{CPU}} = \{\text{CPU\_}\texttt{ix86}, \text{CPU\_SPARC}\}$ then the resources offered by $a_1 \in \mathcal{R}_U$ in Figure 3.3(a) would satisfy both c_2 and c_3 defined below, but $a_2 \in \mathcal{R}_U$ in Figure 3.3(b) would satisfy only c_2:

$$u \in \mathsf{require}(c_2, t_f) \iff u(t, \mathcal{R}_{\text{CPU}}) \geq t \;\text{ if } t \in [0, t_f]$$

$$u \in \mathsf{require}(c_3, t_f) \iff u(t, \mathcal{R}_{\text{CPU},\texttt{ix86}}) \geq t \;\text{ if } t \in [0, t_f]$$

This is because contract c_2 requires 1 CPU unit of any type per second, while c_3 requires the same resources on an `ix86` processor only. As a_1 offers resources on an `ix86` processor, and a_2 a SPARC, we obtain the result above. To illustrate this, $a_2 \notin \mathsf{require}(c_3, 5)$ since $a_2(4, \mathcal{R}_{\text{CPU},\texttt{ix86}}) = 0 < 5$ even though $4 \in [0, 5]$.

Formal contract specification ensures that participants can consistently interpret a contract's terms, and evaluate contract compliance. This also allows them to assess the suitability of a contract before entering into it, during contract negotiation and signing.

3.3 Contract Negotiation and Signing Protocols

For computational contracts to be useful, all participants must agree to them. This agreement could be expressed using digital signatures to prove contract acceptance — ultimately, each participant seeks to prove to itself that the other participant understood and agreed to the contract. Formal contract specification ensures a common interpretation, while contract negotiation and signing allow proof of agreement.

Proof of identity is also important in contract signing, to prevent rogue participants from pretending to be others. However, it is not always necessary to bind computational and physical identities together, or to prevent participants from creating many pseudonymous identities. When this binding is needed, dedicated, well-known naming services can be used to verify participants' public keys and hence their identities (as described in Section 2.3). In practice, participants typically need to use consistent identities to establish trust in their contract compliance in any event (see Section 4.1.2), and public/private keys prevent man in the middle attacks for future interactions.

Contract and message signing protocols have been extensively studied elsewhere [36, 39]; they allow contract signatories to prove to others that they accept the terms of a contract. Fairness is an important aspect of contract signing schemes, and is itself part of the more general topic of fair exchange. In a fair exchange protocol, eventually 'either all involved parties obtain their expected items or none (even a part) of the information to be exchanged with respect to the missing items is received' [69]. In contract signing, this means that ultimately either all parties will provably accept the contract, or all will reject it.

Any contract signing scheme trades off caution against speed of response, and accountability against participant anonymity. On one extreme, for every action a message could be signed, countersigned and registered with a trusted third party. Conversely, participants could enter into contracts based on blind trust, with no computational proof of the other party's intentions. Although neither scheme is suitable for all applications, both are used successfully in certain niche applications: for example digital wallets facilitate online payment by credit card by signing payment authorisation messages which the card issuer must countersign before the payment is accepted [31]. On the other hand, many scientific grid applications simply restrict access to their computers to a known set of users or to credentials issued through pre-approved organisations [14].

This section therefore approaches contract signing from an essentially functional perspective, but with the understanding that the outcome of a contract signing is often unknown or unprovable when it is actually needed. This could be because of unreliable communication links in a distributed system, or the result of deception from cheating participants. Thus the contract framework must allow decisions to be made with incomplete information, based on past experiences and risk estimates. These will in turn be based on earlier experiences attested to through the signing process, hence the need for signature fairness to ensure information symmetry, eventually.

Other aspects of fairness are also important in a signature scheme: abuse-freeness [39] and timeliness [69]. Abuse-free protocols prevent participants from ever proving to others that they have the power to force a contract to be accepted or rejected, part way through the exchange; this ensures that participants cannot leverage incomplete signings in negotiating further contracts with others. In contrast, timeliness ensures that no participant can indefinitely delay the progress of a protocol. Thus abuse-freeness and timeliness represent fairness about making protocol statements, and fairness about controlling progress.

If a signature scheme is to be suitable for contract signing and resource exchange, it should interact as little as possible with the contract framework, so that signings can be treated as primitive, lower-level operations. To ensure this, we would like the scheme to have the properties of *fairness* (to ensure mutual information), *abuse-freeness* (to prevent signature leveraging) and *timeliness* (to discourage stalling and hedging). Although this does not prevent signature interactions, it ensures they can only be based on probabilistic reasoning and trust, not on certainty and proof. For example, a manipulative participant would not be able to indefinitely delay the signing of a contract while looking elsewhere for a more favourable alternative. However, they could briefly consider other options, but with some risk of being forced by the signature scheme to agree to enter two contracts at once.

We model the progress of a signature exchange with the `Exchange` programming interface shown below. While a more detailed interface might allow more opportunity to manipulate the signature mechanism, this interface provides enough information and control to monitor and avoid others' signature delays or abuses.

```
interface Exchange {
  complete() // Try to successfully complete the exchange, if possible
  abort()  // Try to abort the exchange, where possible
  ExchangeState getState()  // Return INITIAL, COMPLETED, ABORTED
                            // or UNKNOWN
  addListener(ExchangeListener e)  // Monitor state changes
}
```

For example, the following exchange protocol provides fairness, abuse-freeness and timeliness [3, 69], when A and B exchange an item for a signature. Although the help of a trusted third party (TTP) is needed when recovering from failures in the main protocol, the protocol is optimistic in the sense that successful exchanges need not involve the TTP.

- Main protocol:
    1. $A \rightarrow B$: committed item (the TTP can open it without A's help)
    2. $B \rightarrow A$: committed signature
    3. $A \rightarrow B$: item
    4. $B \rightarrow A$: signature
- Recovery protocol:
    1. A or $B \rightarrow$ TTP: committed item and B's committed signature
    2. TTP $\rightarrow A$: B's signature
    3. TTP $\rightarrow B$: A's item
- Abort protocol:
    1. $A \rightarrow$ TTP: abort request
    2. TTP $\rightarrow A$: abort confirmation
    3. TTP $\rightarrow B$: abort confirmation

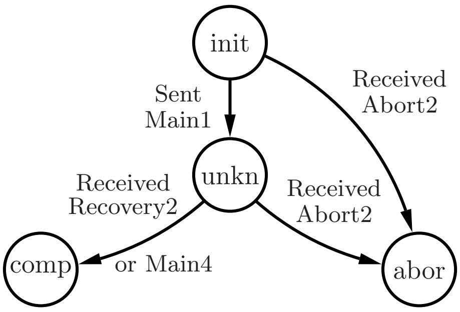


Figure 3.4. A state transition diagram for fair information exchange

The TTP will allow either a recovery or an abort for a particular exchange, but not both. If the main protocol is completed successfully, then a subsequent abort is ignored. Figure 3.4 shows the correlation between protocol steps and signature states reported by `getState()`, from the perspective of the initiating participant, A.

In one sense, it can be argued that this protocol is not fair in conventional terms [69]; for example, B is informed by the TTP if A aborts the protocol. On the other hand, B can effectively abort without informing A, by not replying to the first message. However, these issues affect only the inner workings of the exchange protocol, not the outcomes. The fairness of the exchange is still preserved, thus participants have little incentive to manipulate the protocol, as they are blind to the information being exchanged.

The protocol is also well suited to distributed operation when used for signing and countersigning contract messages, despite the need for trusted third parties. This is because the actual content of a signature is not usually needed except as evidence of good faith dealings. Thus participants can prove to themselves that a signature can be recovered, and defer the actual recovery until the TTP can easily be contacted.

3.3.1 Contract Messages

Signatures on contract messages are important because they are proof of a willing exchange between principals. Firstly, they allow principals to verify the origin and authenticity of each message, preventing attackers from interjecting misinformation. Secondly, signed messages can be used as evidence, demonstrating a participant's interaction history to others as a credential (see Section 4.2).

However, there is more to contract maintenance than simply exchanging signatures. This section outlines how contracts are negotiated and agreed upon, maintained and terminated, between two participants identified by their public/private key pairs. It also illustrates how contracts can fail, and examines the possible consequences.

There are three essential phases to the life-cycle of a contract: initial negotiation, activity, and termination. These phases are controlled by asynchronous messages for requesting actions and exchanging signatures.

Initial negotiation can establish a new contract between two principals, or update the terms of an existing contract. A *contract request* proposes a contract but is not a binding offer. (The request is, however, signed to prove its origin and protect against denial of service attacks.) These requests allow participants to negotiate contracts inexpensively, without having to set aside resources in advance, as they would otherwise have to if the requests were binding.

A *contract exchange* establishes a binding contract, provided both principals agree to its terms by signing it; a copy of the contract together with both signatures proves that it was accepted. Ideally, a fair exchange protocol would be used, ensuring either a *completed contract exchange* in which each participant receives the other's signature, or an *aborted contract exchange* with no signature exchange.[2]

Activity in a contract is regulated through payment messages; participants use these to claim or validate that they have performed their part in a contract, and to request payments from others. As above, there are two forms of payment messages: signed but non binding *payment requests*, and signed *payment exchanges* which become binding if the exchange is completed successfully.

The terms of an active contract can also be modified on the fly, with a new contract exchange. This might be used, for example, to rectify a contract breach such as a missed payment or a resource shortfall.

Termination concludes a contract cleanly, and consists of optional *termination requests* before a final *termination exchange*. Even though a client or server could simply abandon a contract part way through, this would

[2] Without a fair exchange, contracts can still be agreed on using signed and countersigned contract messages, but this would involve some extra risk — for example, when receiving a signed contract, the recipient could delay before countersigning or rejecting it, while using the contract as a bargaining tool with others.

> risk a loss of trust in future contracts, so proper contract termination is important.
>
> Conversely, proof of a successful termination could also be used as evidence of past trustworthiness, to allow participants to bootstrap their trust relationships with others.

This messaging protocol allows contracts to be modified dynamically with the consent of both participants. An alternative, equivalent approach would be to terminate the old contract, and simultaneously begin a new contract linked back to the old one. This could again be used to resolve contractual breaches, but would introduce more race conditions than the approach above; for example, if a client was trying to simultaneously make a contract payment, and upgrade the contract by terminating it, the payment might be accepted or rejected depending on whether it was received before or after termination. If the payment was rejected, it would need to be resent in terms of the new contract, or else the definition of contract identity would need to be changed to avoid this, simply mirroring the message scheme above.

Figure 3.5 illustrates the progress of a typical contract for a computer server application, from initial negotiation through to termination. Each step shows a single higher level activity (such as ‘Contract Request’); for clarity the initial steps also show the lower level messages needed to effect them.

This example is by no means exhaustive. For example in step 1, if the client had considered contract c_2 unsuitable, it could have aborted the signature exchange, and then proposed a third contract, or waited for another offer from the server, or abandoned the negotiation altogether.

The simplicity of the exchange in Figure 3.5 belies the complex decisions that must be made in deciding how to proceed with a contract. For example, which contracts should be accepted? How often should payments be requested? In one sense, these decisions are part of an implementation, not part of the protocol design. Furthermore, they are tightly connected to the trust model, which moderates them in order to manage cost and risk, as discussed in more detail in Section 4.2. As a result, the issues are addressed only briefly below.

In adjusting contract performance, the trust engine must essentially trade off the overheads of sending additional messages and signatures, against the risk of the other party cheating. Since most of these messages will be accounted and charged for under the contract, it would seem at first that servers actually have an incentive to send as many contract messages as possible. However, the

Step	Message flow	Message and contents
Initial negotiation phase:		
0	Client⟶Server	Contract Request for contract c_1=‘xxx’ *Client sends Server* c_1*, and a signature* $sig_{Client}(Request\ c_1)$
1	Server⟷Client	Contract Exchange of contract c_2=‘yyy’ *Server sends Client* c_2*. Then server and client exchange signatures* $sig_{Server}(Exchange\ c_2)$ *and* $sig_{Client}(Exchange\ c_2)$
Activity phase:		
2	Server⟶Client	Payment Request
3	Client⟷Server	Payment Exchange
Termination phase:		
4	Client⟶Server	Termination Request
5	Server⟶Client	Payment Request
6	Client⟷Server	Payment Exchange
7	Server⟷Client	Termination Exchange

Figure 3.5. Example of the messages of a contract

client would then perceive this as an extra overhead for that server, and either choose another server or else appropriately discount the rate at which it was prepared to exchange resources with the server in future. Clients have a similar cost incentive to send as few messages as possible.

On the other hand, especially for long-running contracts, both clients and servers need to ensure each other's compliance, to ensure they are not defrauded of resources. Even messages which cannot be attributed to any successful contract must still be accounted for — e.g. from aborted contract negotiations — to prevent Denial of Service attacks, by adding a dummy contract for overheads. There is even a subtle benefit to the cost overhead of signing messages: participants attempting to attack the system need to consume their own computational resources in signing the messages they send. This makes it more costly to stage an attack, decreasing the rational incentive to do so.[3]

To further protect against local and distributed DoS attacks, proposers of new contracts could also be required to prove their suitability by adding a HashCash [6] stamp to their new contract requests. This proof-of-work stamp is computationally expensive to generate but cheap to check, providing an adjustable barrier of entry for untrusted participants into the contracting framework.

[3] Well-funded, irrational attacks are essentially impossible to guard against; a rational attack expects to gain more resources than it expends, while irrational attacks attempt to cause disruption while expecting a net loss of resources. Thus irrational attacks become indistinguishable from an ordinarily overloaded system.

3.3.2 Contract Selection

Participants can also use the cost of resources to decide which contracts to accept or reject, by rationally comparing real costs against the terms offered. A compute server might know the real cost of its communication bandwidth, charged by its Internet service provider, and could aim to amortise the cost of the computer equipment and annual electricity costs in assessing the cost of providing CPU cycles. In assessing a contract, the server could offset these expected costs against the subjective value of the payment expected.

This subjective economic model lets participants predict which contracts should be worth accepting — but what happens when things go wrong? Again, the economic model provides a mechanism for monitoring contract performance, and assessing this against the contract terms. However, the difficulty is in deciding what to do about this — which depends on correctly identifying the source of the failure:

Server failure The server might fail in its contractual obligations, either deliberately or because of a shortage of resources. Alternatively, the server might try to defraud the client, by misreporting resource usage.

Client failure is similar to server failure; this occurs when the client fails to make a payment expected under a contract.

Contract failure represents an inconsistency between the high-level resource representation of a contract, and the resources needed for its low-level contract actions. This is thus a mismatch between a contract's terms and its actual resource requirements. As a result, performing the contract may require more or fewer resources than expected. Although this is not strictly speaking a failure of the contract, it may cause the contract to fail, even though both client and server have complied with the contract terms.

Unless the contract's terms are renegotiated, neither client nor server can decide from the resource usage alone whether the contract has nearly finished, and is worth completing — even if there are extra resources available.

Communication failure can cause a contract to fail, by masquerading as an apparent server failure from the client's perspective, or vice versa. Thus a contract could fail even though both client and server behaved honestly and correctly.

The cause of failure is not always clear, either because of a lack of information or because of deliberate misinformation. For example, an apparent communication failure could be used by a server to disguise a server failure caused by a shortage of resources.

Failures such as these are inevitable in a large scale distributed system. However, they need not lead to a complete breakdown of the contracts involved. For example, if a server and client persist with a contract, despite communication failures, they may be able to complete it successfully by ignoring occasional late payments — one of the goals of trust modelling is to support and justify this risky behaviour appropriately in a distributed system without encouraging opportunistic attacks on gullible participants. Indeed, even if one or both sides was cheating, a failing contract might still be successful and mutually profitable.

Trust modelling also provides a longer term incentive to perform contracts honestly and correctly, in spite of short-term losses. A market linked contract might become more or less profitable than expected, or a new and more lucrative contract might be available. The only incentive to persevere with the loss-making contract might be the risk of losing others' trust in future.

This chapter has presented a framework for computerised contracts, which allows users (human or electronic) of a computer system to explicitly negotiate their resource needs and terms of payment in terms of a rich resource model. The applications of this framework range from supporting profit-making or co-operative computer services to improving automation of interactive processes by valuing the user's time (see Chapter 5). For this approach to be effective, the resource overheads of monitoring also need to be included, and contract terms must be offset against the trustworthiness or reliability of the participants.

The basic contract model imposes few constraints on resource structure or contract terms, and supports many applications including compute servers which run others' programs for money. Asynchronous contract messages are also defined for negotiating, managing and terminating contracts, leading to an analysis of the causes of contract failures and the need for trust modelling.

Chapter 4

Trust Modelling

Trust modelling completes the contract framework, by providing feedback and control of contract performance — this is necessary to protect against cheats and unreliable participants. Here 'trust' represents the expected disposition or behaviour of a participant in various circumstances. By ordering these trust values, a distinction can be made between suitable and unsuitable contract partners.

This chapter shows how formal modelling of trustworthiness can help guarantee protection from attacks, while encouraging profitable interactions. These properties are proved formally here for a general model of contractual interactions, thus they also cover all implementations which conform to this model.

A trust assessment is necessarily local and subjective, both to prevent self-aggrandising cheats, and because of the nature of distributed systems — for example, an unreliable link between two branch offices of a company might cause participants in each branch to distrust contract offers from the other branch, but trust their local colleagues. This apparently contradictory trust assignment is valid because trustworthiness can depend on the assessor's identity and network location.

Participants can use their trustworthiness to vouch for others, through 'trust recommendations'. These recommendations can help bootstrap the trust system, by providing strangers with a reason to trust each other. Furthermore, recommendations can be used to structure and manage trust relationships; a consulting company could vouch for its employees, to ensure that they were trusted to use clients' computers. Recommendations also subsume trust reputation agencies — conventionally used as initial sources of trust — but in a

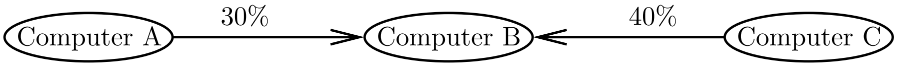


Figure 4.1. A simple trust map

way that is safe and appropriate for distributed systems.

Trust and contracts are bound together by the contract resource model. This provides a concrete interpretation of trust, in the same terms in which contracts are specified. These contracts need to be negotiated honestly and accurately. To ensure this, contract compliance is monitored, and expressed using the trust model. The trust model essentially represents each participant's belief in the expected compliance of every other principal.

The chapter begins by defining trust models in general terms, together with an analysis of the role and usefulness of trust recommendations in trust management. The trust model template is then extended to computational contracts to produce a formal, general purpose trust model for all contracts. This general model provides essential safety and liveness guarantees, which prove that all implementations built around it are safe from attack but allow productive contracts to proceed. Lastly, a typical application of the trust model is shown, in the form of a compute server which sells its computational resources for money. This implementation shows how contracts and trust recommendations can be represented as computer code, and demonstrates how this relates back to the formal model, ensuring its safety.

4.1 Definition of a Trust Model

A trust model represents the trustworthiness of each principal, in the opinion of other principals. Thus each principal associates a trust value with every other principal. For example, Computer A might hold the belief that Computer B successfully completes 30% of contracts, while Computer C might believe that Computer B was 40% successful. These trust values could either be stored explicitly by the trust model, or derived implicitly when needed.

The trust value assignments in this simple example are illustrated in Figure 4.1 as a directed graph. Such a simplistic, linear range of trust values (e.g. 0% to 100%) is generally not enough: a principal's trustworthiness might vary depending on the context or application, being higher in some areas and lower in others. Instead, we define a family of trust models in terms of their external properties, which includes the earlier example as a special case.

We use the following symbols and terms in modelling trust:

Principals $\mathcal{P}$ engage in contracts, which they can digitally sign. They can also sign other statements such as trust recommendations, in order to assert them to others; in practice, a principal $p \in \mathcal{P}$ is identified with its public/ private key pair [72], and the two can be treated synonymously.

As a result, principal identities are pseudonymous: a user or a computer could be associated with any number of principals, and a single principal might apparently operate from more than one location simultaneously — if multiple processes shared copies of its private key.

Although principals represent computational processes, they are often named after the machines, people or organisations for which these processes act, e.g. Computer A, Olivia or Amazon. For illustrative purposes, generic principal names Alice, Bob and Charlie are often used as placeholders for real principal identities.

Trust Values $\mathcal{T}$ represent belief in a principal's trustworthiness. In essence, a single trust value $t \in \mathcal{T}$ represents everything one principal can express about another's trustworthiness, when predicting its actions.

A Trust Model T_m shows each principal's belief in every other principal's trustworthiness, $T_m : \mathcal{P} \times \mathcal{P} \rightarrow \mathcal{T}$. This view of the trust model defines only its external structure; internally it could use other information such as recommendation certificates to derive the trust values.

A wide range of trust models is possible, from static lookup tables provided by centralized reputation agencies (as used by some credit card machines) to local, subjective models which can take into account personal recommendations and certificate-based evidence of trustworthiness.

The rest of this section refines the overall structure of the trust models we use, in terms of how trust values are ordered and calculated. These are then applied to the contracting framework, which allows us to establish a general purpose trust model for contracts.

4.1.1 Trust Ordering

Trust values can often be compared; in the simple example above, Alice might consider Bob less trustworthy than Charlie does. However, not all trust value pairs are comparable: for example Alice might rate Bob's trustworthiness in

	Alice	Bob	Charlie
Alice	100%	30%	30%
Bob	0%	100%	0%
Charlie	10%	40%	90%

Table 4.1. Table of simple trust assignments

one area more highly than Charlie does, but do the reverse in another context. In that case, Alice and Charlie's trust values for Bob might be incomparable. Thus 'trustworthiness' forms a partial ordering of trust values, denoted $\preceq$.

We extend this to a complete partial order (CPO) by assuming that every set of trust values $T \subseteq \mathcal{T}$ has a greatest lower bound. This allows trust values to be combined safely. In particular, this means that there is a unique minimal trust value $\perp_{\preceq}$, the bottom trust element: this represents the worst possible trustworthiness a principal can have.

In many applications, another special trust value is identified, representing an unknown principal $t_{unknown}$. This is particularly useful for implementing a trust model, where the space of principals $\mathcal{P}$ could be arbitrarily large, but at any given instant each principal is aware of only a relatively small number of other principals. By using the unknown trust value as the default, the trust model can be treated as a sparse matrix for efficiency, storing only the entries that differ from the default.

For example, for the simplistic scenario illustrated in Figure 4.1, the trust model would assign to each participant pair ($\mathcal{P} = \{\text{Alice}, \text{Bob}, \text{Charlie}\}$) a trust ranking from the space of trust values $\mathcal{T} = [0\%, 100\%]$. Here the bottom trust value is $\perp_{\preceq} = 0\%$ and the natural ordering is $< = \preceq$. Suitable trust assignments T_m are shown in Table 4.1.

This illustrates how trust assignments need not be consistent or symmetrical: $T_m(\text{Alice}, \text{Bob}) = 30\%$ but $T_m(\text{Charlie}, \text{Bob}) = 40\%$, and similarly $T_m(\text{Charlie}, \text{Bob}) \neq T_m(\text{Bob}, \text{Charlie})$. However, this model is not expressive enough for some purposes, such as explicitly distinguishing between unknown and untrusted participants; whatever the value of $t_{unknown}$, say 50%, there would be no way to distinguish between a previously unknown participant, and a participant that had demonstrated 50% reliability over hundreds of interactions.

How complex should the trust model be, then? On the two extremes lie trivial trust models, which trust everyone equally, and evidence based models which simply store a complete interaction history. The best solution depends on the application, and must offset the benefits of summarising and precomputing trust values, against the corresponding loss of information. In the trivial example above, this loss of information means that there is not expressive power to simultaneously capture participants' dispositional trustworthiness as a percentage, and also the weight of evidence for this — hence the ambiguity of '50% trust'. This linear scale would be enough for certain simpler representations though, such as a binary good/bad dispositional trustworthiness for which each item of positive evidence could be offset by a corresponding item of negative evidence; this form of belief representation is used for PBIL stochastic search [81] to identify bit values for an optimal solution, and is also implicitly used for simple trust modelling in ad hoc networking [63] and wireless gaming [45] applications.

Extending the trivial model to $\mathcal{T}' = \mathbb{N}^2$ allows successes and failures to be counted independently, providing a much richer representation. Although an instantaneous decision might not need this extra information, the advantage lies in being better able to incorporate history when *updating* trust values. This new model also makes explicit the ambiguity of comparing trust values. For example, if Alice had had 3 successful interactions with Bob, and 7 failures, this might be represented in the trust model as $T'_m(\text{Alice}, \text{Bob}) = (3, 7)$, corresponding to the '30% trust' assessment of Figure 4.1. Similarly, Charlie's 40% trust in Bob might be based on five interactions, with $T'_m(\text{Charlie}, \text{Bob}) = (2, 3)$. How should these trust values be compared? They could be ordered consistently with the trivial example:

$$\begin{aligned}(a_1, b_1) \preceq (a_2, b_2) \iff\ & a_1(a_2 + b_2) < a_2(a_1 + b_1) \\ & \text{or } (a_1 = a_2) \wedge (b_1 = b_2)\end{aligned} \tag{4.1}$$

This is a proper partial order, since it satisfies the properties of reflexivity, transitivity and antisymmetry. Other orderings are also possible, such as an informational ordering $\sqsubseteq$ which identifies which pieces of information could have resulted from the same sequence of observations:

$$(a_1, b_1) \sqsubseteq (a_2, b_2) \iff (a_1 \leq a_2) \wedge (b_1 \leq b_2) \tag{4.2}$$

The ideal trust model is an elusive concept, because of the competing goals it would have to satisfy. In part, this stems from the colloquial use of the word 'trust', to fluidly encapsulate belief, predicted actions, disposition and evidence, as well as other concepts such as physical reliability. Practically, too, a trust model must trade off storing and summarising information. One approach would simply store all accumulated evidence as the trust value for each participant; if the space of evidence were $\mathcal{E}$ then the trust values would consist of all evidence combinations $t \in \mathbb{P}(\mathcal{E}) = \mathcal{T}$. These trust values could be ordered according to information content

$$t_1 \sqsubseteq t_2 \iff t_1 \subseteq t_2 \tag{4.3}$$

with bottom element $\perp_{\sqsubseteq} = \emptyset$ representing no evidence at all. This evidence-based model is simple to use and update, but provides no insight into how trust values should be used to make decisions. These decisions could also be arbitrarily expensive to compute, because the evidence set could be arbitrarily large, making this model doubly inappropriate for monitoring resource contracts.

The opposite approach would be a trust model which summarised information until it was simply a table of decisions for particular questions. The difficulty here is that it becomes impossible to update the trust model directly with new evidence, using only the past trust value. Therefore the ideal trust model needs to bridge the gap between evidence and decisions, allowing new evidence to be easily incorporated into trust values, and allowing the trust values to lead naturally to decisions.

4.1.2 Trust Recommendations

Recommendations allow participants to transfer their trust assessments to others. In isolation, trust values are a useful approach for consistently summarising evidence about participants' past interactions, in order to control future behaviour and prevent abuse of the underlying contract framework. Their use can be considerably extended though, by allowing trust values to be combined together to take into account the opinions of others.

This transfer of trust could be managed either actively or passively. On the one hand, participants could explicitly notify others of whom they trust, and to what extent; this would give them direct control over the trust recommendations they issued (although these might then be passed on to others without their knowledge). These trust recommendations could be seen as promises, vouching for other participants' trustworthiness — so participants who make incorrect recommendations stand to lose trust, and vice versa.

Conversely, participants could passively reveal their trust beliefs to others, to be used or ignored at will. However, as they would not be compelled to use these trust assignments internally, the external beliefs could be assigned any trust values, and could even be changed for different viewers. Furthermore, if a participant found another's trust assignments to be incorrect, then they should be less inclined to trust those recommendations in future, resulting in a loss of trustworthiness, even if the recommendations nominally had no underlying promise. Thus active and passive recommendation models are functionally equivalent, and differ only in perspective. Here trust recommendations are discussed from an active perspective, since this makes explicit the risk of losing trust, and because this message-oriented model is most appropriate for factorising trust models into independent components, suitable for asynchronous distribution over a network.

A recommendation is therefore seen as a statement by a *witness* about the *recommended trustworthiness* of its *subject* principal, to be interpreted by the *receiver*. The witness would sign the recommendation to make it self-contained and unforgeable.

Recommendations can also be seen as credentials, which can be used to obtain trust from other principals, comparable to Rôle Based Access Control (RBAC) credentials which are used to obtain privileges. Trust recommendations can be considered as an extension of the contract framework, by extending the resource model to incorporate trust as another resource. Chapter 5 discusses this and other resource extensions in more detail.

On a more practical level, trust recommendations share many common properties with conventional access control credentials. As with other credentials,

- a trust recommendation is signed by the issuer to certify its authenticity,
- it typically carries a validity period or expiry date to limit its use,[1] and
- it is parameterized or labelled to control the context in which it may be used, allowing only partial access to the space of all privileges.

Trust recommendations not only have a similar representation to RBAC credentials; they are also interpreted similarly:

- the interpretation may be subjective, with different participants interpreting the same credentials differently,

[1] A simple message timestamp would not be enough, since this would provide an ambiguous interpretation affecting the witness's perceived trustworthiness; instead, any validity period would need to expressed in absolute terms.

- unexpired credentials may be taken at face value, revalidated with the issuer or checked against certificate revocation lists, in a trade-off between speed and safety, and
- credentials may be combined or chained together to authorise extra privileges which would not be available if the credentials were presented singly.

Finally, while access control systems use explicit policies to define the privileges associated with credentials, for trust recommendations this is the task of the trust model, which defines how trust values are decided on and combined.

Recommendations allow partial transfer of trust from witnesses to subjects, from the perspectives of the receivers. The trustworthiness recommended by a witness is then *discounted* by the receiver according to the receiver's trust in the witness and any other terms of the recommendation, analogously to the discounting of second hand evidence in logics of uncertainty [52]. This trust transfer necessarily acts in both directions; if a recommendation can change the apparent trustworthiness of its subject, then the subject's actions will also affect the recipient's trust in the witness of the recommendation.

This distributed trust model, based on recommendations, has many advantages over the alternative centralized or ad hoc models. In a centralized model, reputation agencies are used to accumulate and disseminate information about participant trustworthiness. Thus, to determine if a previously unknown principal is to be trusted, one can ask the reputation agency, which acts similarly to a commercial credit rating agency. Although for many small or localized applications this may be a good approach, it is not general enough for distributed applications where participants have limited or unreliable network connectivity, or need to generate and manage new identities dynamically.

Purely local, ad hoc trust models are also inappropriate, since they have no shared terms of reference, and do not allow participants to exchange trust information effectively with each other. As a result, although these models do not suffer the reliability problems of a centralized model, they lose the advantages of having a large trust network in which observations have a common interpretation and can be pooled.

Figure 4.2 shows the basic structure of a typical trust recommendation; the interaction with contracts is discussed in more detail in Section 4.2. In this recommendation, Alice vouches for Bob's trustworthiness in a certain context. The degree of trust shows how much trust Alice says she has in Bob — £3 worth of successful interactions and £7 worth of failures — while the limits show the

Alice recommends Bob	
Context	for contracts within cl.cam.ac.uk
Degree	$t = (3, 7)$ (measured in £)
Limits	£2.00, 25%
Expiry	22 October 2003 at 11:30 UTC
Signature	Alice's signature

Figure 4.2. Example of a trust recommendation certificate

maximum sum and the maximum percentage of a debt Alice is prepared to pay on Bob's behalf. Finally, the expiry date and Alice's signature complete the trustworthiness certificate.

Recommendations are well suited for distributed applications because they need no central administration, they are easily factorised, and they can help take advantage of local broadcast communication mechanisms. Any participant can issue recommendations, so commonly agreed reputation agencies are not required. Still, certain very well known companies such as credit card companies might act as *de facto* reputation agencies when they issued trust certificates; in this sense, the recommendation-based model subsumes conventional centralized models, while still allowing personal recommendations between participants. Because recommendations are inherently self-contained, they can be distributed through a network independently and even asynchronously — for example using gossip protocols [55]. For many applications the small world property holds in which most participants are connected by only a few degrees of separation. For example Alice might not know Charlie, but if Alice knows Bob and Bob knows Charlie then Alice is connected to Charlie with degree 2. In human interactions, this degree has been shown to be typically between 5 and 7 in large-scale tests [27], so if participants store partial recommendation chains starting or ending with themselves, and assume that local broadcast recipients are particularly likely to interact in similar contexts, then participants can efficiently and automatically discover recommendation chains which link them together in order to establish trust.

Chains of recommendations can also be managed manually; Figure 4.3 illustrates a corporate scenario in which a company issues trust recommendation certificates to its employees, so that clients which trust the company will also trust the employees.

Bootstrapping of the trust framework is essential in order that contracts can be established satisfactorily, which will then generate further recommendations automatically. Here recommendations can help too, both directly and indirectly.

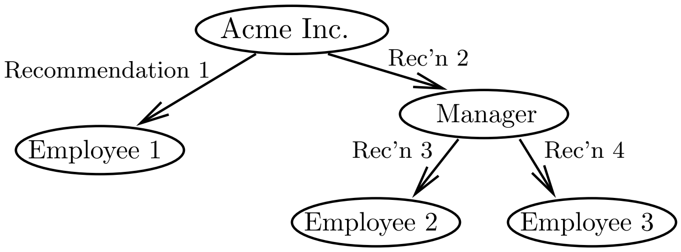


Figure 4.3. Recommendations can give trust to employees

Firstly, recommendations could be issued directly by preconfigured trusted participants; each client would then need to trust only a single preconfigured participant to enter the trust network sensibly, and establish itself. Secondly, a participant could establish a single, imaginary entity to represent unknown principals. By automatically issuing recommendations from this principal to previously unknown participants, new participants would be able to gain some trust initially, which they could then build up through further interactions — provided that the representative of the unknown principals had a good trust value already.

This mechanism would be self-limiting, since over time the trust in the representative would reflect the typical trust of unknown principals. Thus if unknown principals were usually trustworthy then their representative would be trusted, and *vice versa*. Clearly, an attacker could potentially gain resources by faking a number of new identities, to effectively cash in on the goodwill generated by others. However, this effect could be lessened by using the HashCash scheme described in Section 3.3.1 to force the attacker to expend significant resources itself when creating new identities.

A server could also deliberately encourage new participants, banking on future business in order to avoid the overhead costs of standing idle, by effectively donating resources to this by trickle-charging the trust value of the unknown principals' representative. Recommendations are thus an important tool for both bootstrapping and maintaining the trust framework.

The use and chaining together of recommendations does make implicit assumptions about the independence of principal trustworthiness. When the trustworthiness of a principal is assessed, it could in fact be a number of distinct entities working together, such as a web server together with its database on another machine, or a computer and its network. Ideally, the reliability of these components would be modelled independently, by treating each as a principal with its own trust value assignment. If this distinction is impossible to make,

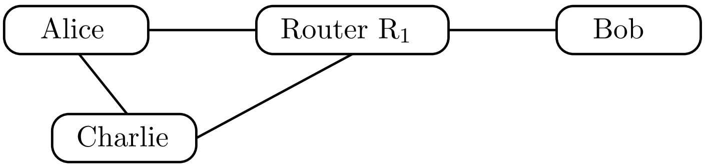


Figure 4.4. Multiple entities can masquerade as a single principal

then these principals will need to be treated as a single compound principal — even if some of the components lie beyond the control of the principal nominally being assessed.

The following discussion demonstrates informally that this is a safe assumption even when recommendations are chained, although it may unfairly weaken the trust in the combination. We assume that observed trustworthiness is decreased by each additional principal which contributes to a route. For example in Figure 4.4, Alice's observations of 'Bob' might in fact be observations of the compound principal $\mathrm{Bob}_{\mathrm{Alice}} = \{\mathrm{Router\ R}_1, \mathrm{Bob}\}$. If Bob's observations of Charlie are also affected by router R_1 ($\mathrm{Charlie}_{\mathrm{Bob}} = \{\mathrm{Router\ R}_1, \mathrm{Charlie}\}$), but Alice is on the same network segment as Charlie ($\mathrm{Charlie}_{\mathrm{Alice}} = \{\mathrm{Charlie}\}$), then Alice might take into account Router R_1's unreliability twice in computing her trust in Charlie via Bob's recommendation, when she should not count it at all. However, because Router R_1's contribution would only serve to decrease the strength of the trust, the resulting trust given to Charlie will be safe, if possibly erring on the side of caution. This result applies similarly when longer recommendation sequences are combined too.

Groups attacks are also effectively protected against using this distributed recommendation model. As long as all resources are accounted for within the contract framework, any attack which affects service performance will be reflected there and identified with the attacker; this will result in a loss of trust for them and their recommenders (up to the recommendation limits). Even when a group of principals attack a server, they can only gain a bounded quantity of resources before they and possibly their clique of recommenders are seen as unprofitable and disallowed access. If a group attacked many servers simultaneously, they could increase their gains in the short term, but only until updated recommendation information was propagated between servers according to the implementation's communication policy. Thus localised, subjective cost assessments and the distributed recommendation model still give appropriate, distributed protection against group attack.

Recommendations may be used in many ways in managing a trust model. But finally, the ontological question remains: what do they really mean? On a practical level, they can simply be seen as tools for transferring trust between principals, akin to RBAC credentials. (Chapter 5 reinterprets this metaphorical transfer as a literal exchange of commodities.) On a second level, recommendations signify both a claim and a promise: a claim that another principal is trustworthy, possibly backed up by explicit evidence of successful contracts in the past, and a promise by the recommender to stand by the claim either by explicitly standing surety for potential debts or by risking a loss of trust if the debts are not repaid.[2]

The essential question is then about the nature of this promise: is it part of the trust framework or does it exist at a higher level? In principal, either answer to this question is a valid implementation decision. However, if trust in a principal's ability to recommend were independent of trust in their ability to perform contracts [50], then higher level recommendations could be made about others' abilities to recommend at successively higher levels, unless this were explicitly disallowed or an arbitrary cap were imposed. Furthermore, the trust levels would not be truly independent, as principals with high recommendation trust could simply recommend themselves (or their pseudonyms) for ordinary trust, in order to attack the system.

Therefore we choose the simpler solution, which equates the ability to perform contracts and the ability to recommend. As a side effect, this makes recommendations into a form of contract themselves, in which the exchange is of trust. This unification of contracts and recommendations shows both the expressiveness of the contract framework (Chapter 3), and the need for a careful analysis of the space of resources (see Chapter 5). Until then, though, we can still practically manage recommendations and their significance as promises, even without explicitly acknowledging them to be contracts. To do this, we first examine the direct effects of trust on contracts and resources, and the resulting properties of safety and liveness which ensure that cheats are ostracised while successful contracts proceed.

4.2 Trust and Contracts

Trust and contracts are bound together by the contract resource model. This binding allows principals' trustworthiness to be computed and monitored as

[2] Principals are not forced to issue recommendations, but those that do will be judged on them.

contracts are performed, and facilitates the selection of appropriate contracts in future based on participants' trustworthiness.

Contracts define an exchange of resources, which the trust model seeks to monitor. The aim of the trust model is to predict how the other principals will comply with their contracts' terms, to help the owner of the trust model conduct profitable contracts. For clarity, this is presented from the perspective of a server offering resources to a client, but the client would assess the server in exactly the same way.

To do this, the server needs to appropriately model causes of failure as outlined in Section 3.3.2, particularly client failure and contract failure. If communication failures cannot be assessed independently then these can be attributed to client failure for simplicity, while server failure (whether deliberate or caused by resource shortage) can also be modelled by the same mechanisms that model client failure, so the server can discount its own reliability appropriately too.

Client failure occurs when a principal does not provide the resources expected by the server, as specified in the contract accounting function. Here the server tries to model how the actual payment compares to the expected payment. While in principle each resource type could be assessed separately, it is simpler for the server to instead assess the subjective value of each payment or resource transfer, against the expected payment's value. Thus the server stores a pair of numbers for each participant (total value of resources received from client, total value expected from client) which can be seen as representing the server's trust in the client in the context of repaying debts. This allows the server to weight the client's contract payment assertions appropriately, to better predict how subjectively profitable its contracts with that client will be.

In contrast, contract failure represents a contract which incorrectly estimates the resources it requires, resulting in a resource shortage. This too can be modelled as a pair of values: (value of actual resource outlay, value of expected outlay). These values could be associated either with the client or with the contract terms, but our assumption is that the client chose the contract's terms and is therefore responsible for the contract. Put another way, if two different clients enter into ostensibly the same contract (but with different input data sets), then one client's faulty resource estimate should have no bearing on our assessment of the other's.

Putting these values together shows that each principal has a 4 component trust value $t \in \mathcal{T} = \mathbb{R}^4$ with

$$\begin{aligned} t &= (\text{expected receipt}, \text{actual receipt}, \text{expected outlay}, \text{actual outlay}) \\ &= (t_{er}, t_{ar}, t_{eo}, t_{ao}) \qquad (4.4) \end{aligned}$$

where each component represents the subjective value of the resources involved. Participants can then use these trust values to adjust their expectations of profitability for future contracts. (These trust values conceptually represent the total accumulated benefits and costs of past interactions with the other participant.)

In the simplest case, if a new client presents the same contract twice in succession, all else being equal, then the server could scale the second contract's expected outlay by $\frac{t_{ao}}{t_{eo}}$ and expected receipt by $\frac{t_{ar}}{t_{er}}$ to get an accurate prediction of its expected subjective profitability.

The net profitability of a client from a server's perspective can also be assessed from this trust value, as $t_{ar} - t_{ao}$. This can be used to help decide on the total resource outlay that should be allowed before demanding payment from the other party in a contract. The implicit assumption is that this historical profitability can be treated as a reserve, and tapped into for future contracts, as long as the net profitability remains positive. This has important implications for profit-taking by both participants in a contract. From the perspective of a server holding a client's trust quadruple (or vice versa), it may be necessary to reduce the risk associated with this profitability reserve, and set aside a portion of it in case the client is compromised or cheats in future. This can be achieved either by scaling down all components of the client's trust value proportionally (assuming that client trustworthiness has remained the same over time, perturbed only by measurement errors), or by discarding the contribution of earlier contracts to the totals (assuming clients become more or less trustworthy over time).

Profits can also be taken from the other side of a contract too, although with less predictable consequences. For example, a client may try to induce a server to continue to give it resources, while delaying any payments. However, the client cannot know when it will exhaust the server's patience, making it difficult to predictably obtain resources in this way. Furthermore, this client profit-taking could jeopardise future contracts too, as it reduces the client's apparent trustworthiness for the server. Finally, this increases the overheads of contracts, such as the extra resources consumed in supporting more frequent payments.

Thus profit-taking decreases the risk of loss if behaviour changes in future, but also decreases the profitability of future contracts, if past behaviour is continued.

The trust model's profitability assessment shown above has many similarities with the successful 'tit for tat' strategy for the iterated prisoner's dilemma problem (see Section 2.2). In both strategies, each participant tries to compensate for the other's behaviour by mirroring their actions. In tit for tat, cooperative behaviour is rewarded with cooperation in the next iteration, while the trust model effectively inflates prices when the other party underpays. Still, there are significant differences too, because the contract framework has contracts with a wide range of sizes, and because contract selection is not an objective decision because of the subjectivity of assessing profitability.

Contracts have arbitrary sizes, in terms of the value of the resources involved. As a result, their effect on trustworthiness must also be scaled proportionately — otherwise a client could undertake many small, low-valued contracts in order to gain extra resources later from a single very large contract. Section 4.2.3 shows more formally that it is impossible to consistently pump resources from the system in this way. In contrast in the prisoner's dilemma, all interactions are equally significant in potential value. This has two side effects: firstly, the tit for tat strategy needs only a single item of history to decide on its next action, because past slights can be made up for in a single step. Secondly, constant transaction sizes in the prisoner's dilemma, and no choice in selecting the other party, mean there is no notion of relative risk in tit for tat, again negating the need to store extra state.

Continuing the analogy, when measurement error is incorporated into the prisoner's dilemma, tit for tat is often outperformed by a generous variation [46], in which cheating is occasionally forgiven. This can be compared to the strategy described in Section 4.1.2 of trickle-charging principals' trust as a system overhead, for bootstrapping and to encourage a high system load.

In this way, the trust model seeks to ensure that contracts are profitable, by monitoring expected and actual resource usage, and using these to predict the resource needs of future contracts. This general purpose model is specified formally below, and then proved to give important practical guarantees of liveness and safety.

4.2.1 General Purpose Trust Model for Contracts

By basing trust for contracts on resources and profitability, we have established a formal, measurable definition of trust that is practical yet simple to use. It is based on the following assumptions:

- Contracts break down because of a shortage of resources (including payments) or because they were incorrectly specified.
- Participants are expected to make similar errors in payment and prediction in future to those they made in the past.

  (Other mechanisms such as profit-taking and intrusion detection can instantaneously change trust assessments, effectively rewriting history, but these act outside of the trust model.)
- Profitable contracts from profitable clients are to be preferred, based on a local assessment of the costs of the resources involved.

The contract trust model for use by a principal p operates in the following way: Let $\mathcal{P}$ be the space of all principals.[3] Then p stores its trust beliefs using

$$\begin{gathered} T_m : \mathcal{P} \times \mathcal{P} \to \mathcal{T} \qquad (4.5) \\ \text{with } \mathcal{T} = \mathbb{R}^4 \text{ and } t = (t_{er}, t_{ar}, t_{eo}, t_{ao}),\ \forall t \in \mathcal{T} \end{gathered}$$

When p decides whether to interact with principal p_1 on a contract c_1, it needs to consider three factors: whether interactions with p_1 are profitable, whether the contract offers good returns, and whether p has the resources to support the contract. This section shows how these calculations are performed.

1. To assess participant profitability, let

   $$(t_{er}, t_{ar}, t_{eo}, t_{ao}) = T_m(p, p_1) \qquad (4.6)$$

   Reject the contract if $t_{ar} - t_{ao} \leq 0$, since this would show p_1 to be unprofitable.

2. Compute the expected return on investment (ROI) for the contract subjectively. (This is possible only if p has past experience of p_1, i.e. $t_{er} \neq 0$,

[3] Although $\mathcal{P}$ could be arbitrarily large, principals need only consider those other principals with which they have interacted, or whose recommendations they accept.

$t_{eo} \neq 0$.)

$$\text{Expected ROI} = \frac{\text{Cost of expected return for contract } c_1}{\text{Cost of expected outlay for contract } c_1} \tag{4.7}$$

p computes this by deriving a second contract c_1' which adjusts c_1 according to p's trust in p_1, by scaling the expected resource usage by $\frac{t_{ao}}{t_{eo}}$ and the accounting function by $\frac{t_{ar}}{t_{er}}$. In the formalism of Section 3.2, this means

$$\mathsf{require}(c_1', t) = \left\{ \frac{t_{ao}}{t_{eo}}.v,\ \forall v \in \mathsf{require}(c_1, t) \right\} \tag{4.8}$$

with scalar multiplication naturally defined. Similarly

$$\mathsf{account}(c_1')(u) = \frac{t_{ar}}{t_{er}}.\mathsf{account}(c_1)(u),\ \forall u \in \mathcal{R}_U \tag{4.9}$$

Then the cost of the expected outlay is computed by applying p's resource cost function cost_p to the resource requirements. As the requirements are formally expressed as a set of acceptable resource combinations, there may be a choice of different resource outlays which satisfy the contract, from which p selects the one which matches the resources it expects to offer, in order to perform an accurate cost analysis. For example, if p always tries to choose the least costly outlay, the expected resource outlay will be

$$v_{best} \in \lim_{t\to\infty} \mathsf{require}(c_1', t) : \text{cost}_p(v_{best}) = \inf_{v \in \lim_{t\to\infty} \mathsf{require}(c_1', t)} \text{cost}_p(v) \tag{4.10}$$

$$\text{Expected ROI for } c_1 = \frac{\text{cost}_p(\mathsf{account}(c_1')(v_{best}))}{\text{cost}_p(v_{best})} \tag{4.11}$$

(This cost infimum is guaranteed to exist as long c_1 is satisfiable with some resource combination, since costs are real valued. Technically, it might be that no minimum cost v_{best} existed, but in that case there would be other resource combinations arbitrarily close in cost to v_{best} which did satisfy contract c_1, which could be used instead.)

Contracts are then prioritised according to expected ROI; those with ROI ≤ 1 are rejected automatically as unprofitable, and those with the highest ROI are considered first.

3. Finally, p considers whether it has the resources available to accept a particular contract: either the v_{best} calculated above in Equation 4.10, or some other acceptable resource combination (although this would then require a recalculation of the expected ROI).

4. Once the contract has been accepted it will be executed, constantly updating p's trust in p_1 as resources are used and received. If ever $t_{ar} - t_{ao} \leq 0$, the contract may be suspended until more resources are received.[4] Otherwise, the contract continues until it is completed.

Adding Recommendations to the Trust Model

Recommendations can also be incorporated into the general purpose trust model above. As with the rest of the trust model, this still leaves scope for configuring exactly how recommendations are interpreted for a specific application. However, this model ensures that all implementations, however configured, will nevertheless satisfy the essential properties of safety and liveness proved below.

One interpretation of a recommendation is as an actual contract itself, a promise to provide resources on behalf of another principal. Together with this, there would also be a reciprocal contract rewarding the recommender with a dividend if their recommendation was helpful.

Interpreting recommendations explicitly as contracts would require an extra level of management in the contracting framework, as discussed in Section 4.1.2. Instead, they can be directly integrated into the trust model as follows:

Let $\mathcal{C}_{rec}$ be the space of possible recommendations. (We consider $\mathcal{C}_{rec}$ to be a subspace of the space of all contracts $\mathcal{C}$.) Assume principal p has an initial trust model T_m and a collection of recommendations $Rec \subseteq \mathcal{C}_{rec}$. Then, after the recommendations have been considered, the effective trust values governing a contract c_1 are defined by

$$\mathsf{consideredTrust}(T_m, Rec, c_1) = T_{m,considered} \tag{4.12}$$

This function defines how recommendations lead to trust values, which are then interpreted as before according to the algorithm given above.

The way in which the trust model is updated also changes when recommendations are taken into account. Without recommendations, individual trust entries are updated as a contract is processed by simply adding the extra resources expected, used or received. However, when recommendations are included, a function of the following form is used to define how updates change trust values:

[4] If the profitability $t_{ar} - t_{ao}$ falls below a certain lower bound, the contract could be cancelled instead of suspended, avoiding any extra resource cost.

$$\mathsf{updatedTrust}(T_m, Rec, p_1, c_1, \Delta t, p_2) = (T'_m, Rec') \tag{4.13}$$

where p_1 is the other participant in the contract c_1, $\Delta t \in \mathcal{T}$ is the change in resource value, and p_2 is the participant which actually contributed the resources.

Finally, any valid recommendation framework must preserve trust model safety, by ensuring that trust is conserved when trust values are updated (Equation 4.14) and that suspended principals cannot give trust to others through recommendations (Equation 4.15):

$$\begin{aligned} \text{If } \mathsf{updatedTrust}(T_m, Rec, p_1, c_1, \Delta t, p_2) &= (T'_m, Rec') \\ \mathsf{consideredTrust}(T_m, Rec, c_1) &= T_{m,considered} \\ \mathsf{consideredTrust}(T'_m, Rec, c_1) &= T'_{m,considered} \end{aligned}$$

$$\text{then } \sum_{p_3 \in \mathcal{P}} T'_{m,considered}(p, p_3) = \sum_{p_3 \in \mathcal{P}} T_{m,considered}(p, p_3) + \Delta t \tag{4.14}$$

and

$$\begin{gathered} \forall p_3 \in \mathcal{P} \setminus \{p_1\}, \text{ if } (t_{er}, t_{ar}, t_{eo}, t_{ao}) = T_{m,considered}(p, p_3) \text{ and} \\ (t'_{er}, t'_{ar}, t'_{eo}, t'_{ao}) = T'_{m,considered}(p, p_3) \\ \text{then } t'_{ar} - t'_{ao} \geq 0 \text{ or } (t_{ar} - t_{ao} \leq 0 \text{ and } t'_{ar} - t'_{ao} \geq t_{ar} - t_{ao}) \end{gathered} \tag{4.15}$$

Together, $\mathsf{consideredTrust}$ and $\mathsf{updatedTrust}$ allow trust values to take into account both general recommendations and recommendations which apply only to particular contracts. Furthermore, resource usage can change not only trust in the participant directly involved, but also trust in others and the recommendation state. This might be used, for example, if a particular recommendation had a maximum liability limit, to allow incremental resource usage to be counted cumulatively towards the limit.

The original trust model, without recommendations, is in fact a special case of this recommendation model, as shown by the following definitions of $\mathsf{consideredTrust}_0$ and $\mathsf{updatedTrust}_0$:

$$\begin{aligned} \text{Let } \mathsf{consideredTrust}_0(T_m, Rec, c_1) &= T_m \\ \mathsf{updatedTrust}_0(T_m, Rec, p_1, c_1, \Delta t, p_2) &= (T'_m, Rec) \\ \text{where } \forall p_3, p_4 \in \mathcal{P},\ T'_m(p_3, p_4) &= \begin{cases} T_m(p_3, p_4) + \Delta t & \text{if } (p_3, p_4) = (p, p_1) \\ T_m(p_3, p_4) & \text{otherwise} \end{cases} \end{aligned}$$

Clearly, these functions $\mathsf{consideredTrust}_0$ and $\mathsf{updatedTrust}_0$ satisfy Equations 4.14 and 4.15, showing that trust is effectively conserved and suspension is respected.

A more sophisticated recommendation model is given in Section 4.3 for a compute server application, which also satisfies the same constraints. This illustrates how minimal restrictions in the general purpose trust model can support a wide range of applications, without needing to impose excessive simplicity or unnecessary complexity on all implementations.

When complex interconnected recommendations need to be combined, the algorithm which calculates the resulting considered trust values should be shown to be consistent, so that the better a participant's initial trust, the better their resulting trust. If this function is suitably monotone, then the algorithm for calculating this trust can be re-expressed as the function's least fixed point [107]; Section 5.3 illustrates trust assignments with recommendations in these terms.

Furthermore, the general purpose operational trust model defined above ensures useful properties of liveness and safety, which show that the trust model continues to accept profitable contracts and is protected against systematic resource theft.

4.2.2 Trust Model Liveness

Theorem 4.1. *Correct, profitable contracts remain subjectively profitable when correctly performed.*

Assume principal p is interacting with principal p_1, who has behaved exactly according to contract to date. If contract c_1 is accepted by p with expected ROI $r_1 > 1$, and if p and p_1 behave honestly and the contract proceeds as stated (and without intermediate errors) then a similar contract c_2 will also have expected ROI $r_2 = r_1 > 1$, as long as the resource valuations do not change over time, and p has similar resources available when considering c_1 and c_2. Furthermore, the contract c_1 increases p_1's overall profitability for p.

Proof: Let T_m be the initial state of the trust model, and T'_m the state after contract c_1 is completed.

$$\text{Let } (t_{er}, t_{ar}, t_{eo}, t_{ao}) = T_m(p, p_1) \text{ and } (t'_{er}, t'_{ar}, t'_{eo}, t'_{ao}) = T'_m(p, p_1)$$

Because p_1 has behaved according to contract to date, $t_{er} = t_{ar}$ and $t_{eo} = t_{ao}$.

$$\text{Now, } r_1 = \frac{\text{cost}_p(\mathsf{account}(c'_1)(v_{best}))}{\text{cost}_p(v_{best})} = \frac{\text{cost}_p(\frac{t_{ar}}{t_{er}}.\mathsf{account}(c_1)(v_{best}))}{\text{cost}_p(v_{best})}$$

where v_{best} satisfies Equation 4.10.

Furthermore, contract c_1 was correct and correctly performed, thus

$$t'_{eo} = t_{eo} + \mathrm{cost}_p(v_{best}) \text{ and } t'_{ao} = t_{ao} + \mathrm{cost}_p(v_{best})$$

$$t'_{er} = t_{er} + r_1.\mathrm{cost}_p(v_{best}) \text{ and } t'_{ar} = t_{ar} + r_1.\mathrm{cost}_p(v_{best})$$

Then $t'_{eo} = t'_{ao}$ and $t'_{er} = t'_{ar}$, and so by Equations 4.8 and 4.10,

$$\mathsf{require}(c'_2, t) = \mathsf{require}(c'_1, t), \forall t \in \mathbb{R} \text{ and } v'_{best} = v_{best}$$

Therefore,

$$\begin{aligned} r_2 &= \frac{\mathrm{cost}_p(\mathsf{account}(c'_2)(v'_{best}))}{\mathrm{cost}_p(v'_{best})} \\ &= \frac{\mathrm{cost}_p(\frac{t'_{ar}}{t'_{er}}.\mathsf{account}(c_2)(v'_{best}))}{\mathrm{cost}_p(v'_{best})} \\ &= \frac{\mathrm{cost}_p(\frac{t_{ar}}{t_{er}}.\mathsf{account}(c_1)(v_{best}))}{\mathrm{cost}_p(v_{best})} \\ &= r_1 \end{aligned}$$

Hence $r_2 = r_1 > 1$, and there is also an increase in profitability since

$$\begin{aligned} t'_{ar} - t'_{ao} &= t_{ar} - t_{ao} + (r_1 - 1).\mathrm{cost}_p(v_{best}) \\ &\geq t_{ar} - t_{ao} > 0 \qquad \text{since } r_1 > 1 \text{ and } c_1 \text{ was initially accepted.} \end{aligned}$$

□

Theorem 4.2. *Correctly predicted contracts are profitable, when performed as expected.*

Assume principal p is interacting with principal p_1. If contract c_1 is accepted by p with expected ROI $r_1 > 1$, and p_1 (and c_1) behave as predicted by the trust model (or better), then a similar contract c_2 will have expected ROI $r_2 \geq r_1 > 1$ and p_1's net profitability $t_{ar} - t_{ao}$ will increase.

Proof: As in the proof of Theorem 4.1, let T_m be the initial state of the trust model, and T'_m the state after contract c_1 is completed, with

$$(t_{er}, t_{ar}, t_{eo}, t_{ao}) = T_m(p, p_1) \text{ and } (t'_{er}, t'_{ar}, t'_{eo}, t'_{ao}) = T'_m(p, p_1)$$

If c_1 proceeds as predicted, then (reversing the scaling of Equation 4.8)

$$t'_{ao} = t_{ao} + \text{cost}_p(v_{best}) \text{ while } t'_{eo} = t_{eo} + \text{cost}_p(\frac{t_{eo}}{t_{ao}}.v_{best})$$

$$\text{Similarly, } t'_{ar} \geq t_{ar} + r_1.\text{cost}_p(v_{best}) \text{ and } t'_{er} = t_{er} + r_1.\frac{t_{er}}{t_{ar}}.\text{cost}_p(v_{best})$$

Then, assuming linearity in p's subjective resource cost function cost_p,

$$\frac{t'_{ao}}{t'_{eo}} = \frac{t_{ao} + \text{cost}_p(v_{best})}{t_{eo} + \text{cost}_p(\frac{t_{eo}}{t_{ao}}.v_{best})} = \frac{1}{\frac{t_{eo}}{t_{ao}}}.\frac{t_{ao} + \text{cost}_p(v_{best})}{t_{ao} + \text{cost}_p(v_{best})} = \frac{t_{ao}}{t_{eo}}$$

$$\text{Similarly, } \frac{t'_{ar}}{t'_{er}} \geq \frac{t_{ar}}{t_{er}} \qquad \text{(provided that } t_{er} > 0)$$

Thus $\mathsf{require}(c'_2, t) = \mathsf{require}(c'_1, t)$, $\forall t \in \mathbb{R}$ and $v'_{best} = v_{best}$.

$$\begin{aligned}
\text{Therefore, } r_2 &= \frac{\text{cost}_p(\mathsf{account}(c'_2)(v'_{best}))}{\text{cost}_p(v'_{best})} \\
&= \frac{\text{cost}_p(\frac{t'_{ar}}{t'_{er}}.\mathsf{account}(c_2)(v'_{best}))}{\text{cost}_p(v'_{best})} \\
&\geq \frac{\text{cost}_p(\frac{t_{ar}}{t_{er}}.\mathsf{account}(c_1)(v_{best}))}{\text{cost}_p(v_{best})} && \text{since } \mathsf{account}(c_1) \\
& && = \mathsf{account}(c_2) \\
&= \frac{\text{cost}_p(\mathsf{account}(c'_1)(v_{best}))}{\text{cost}_p(v_{best})} = r_1
\end{aligned}$$

Furthermore,

$$\begin{aligned}
t'_{ar} - t'_{ao} &\geq t_{ar} + r_1.\text{cost}_p(v_{best}) - t_{ao} - \text{cost}_p(v_{best}) \\
&= t_{ar} - t_{ao} + (r_1 - 1).\text{cost}_p(v_{best}) \\
&\geq t_{ar} - t_{ao} > 0 \qquad \text{since } r_1 > 1 \text{ and } c_1 \text{ was initially accepted.}
\end{aligned}$$

□

4.2.3 Trust Model Safety

Theorem 4.3. *No principal p_1 can extract more resource value from a contract c_1 with p than the sum of its initial profitability ($t_{ar} - t_{ao}$), and the resources it contributed, from p's perspective.*

Proof: This follows as a direct consequence of Step 4 of the trust model definition in Section 4.2.1. Whenever resources are consumed under a contract (beyond some quantisation level), p's current trust model state T'_m is checked to ensure that $t'_{ar} - t'_{ao} > 0$ where $(t'_{er}, t'_{ar}, t'_{eo}, t'_{ao}) = T_m(p, p_1)$, otherwise the trust model is suspended. Provided that all the resources used in these checks are themselves accounted for, and at each step lie below the quantisation level, then p_1 cannot extract further useful resources from the contract.

Let $(t_{er}, t_{ar}, t_{eo}, t_{ao})$ be p's initial trust in p_1. Then the resources value which p_1 contributed is $t'_{ar} - t_{ar}$ and the resource value extracted is $t'_{ao} - t_{ao}$. Then

$$\begin{aligned} & t'_{ar} - t'_{ao} > 0 \\ \iff \quad & t'_{ao} < t'_{ar} \\ \iff \quad & t'_{ao} - t_{ao} < (t_{ar} - t_{ao}) + (t'_{ar} - t_{ar}) \end{aligned}$$

□

Practically, some small quantity of resources might be lost in discovering that a principal is unprofitable, which might not be recovered. However, in practice this loss will be bounded in size, and does not represent profitable work that the attacker could gain by. Furthermore, any loss would be recovered if the contract ever returned to profitability.

Theorem 4.4. *When using recommendations, no group of principals can extract more resource value than they input, combined with their initial considered profitability (or 0 if this is negative), from the recipient's perspective.*

Proof: Although we show this result using a single contract, if concurrent contracts use the same trust model then it will hold for them too. This is because the concurrent contracts could be decomposed into a sequence of contract fragments, which act as a sequence of non-overlapping short contracts from the perspective of the trust model, and to which the proof below applies.

Let $(p_{1,1}, c_{1,1}, \Delta t_1, p_{2,1}), \ldots, (p_{1,n}, c_{1,n}, \Delta t_n, p_{2,n})$ be a sequence of trust updates as contemplated in Equation 4.13 computed by principal p. Let $\text{profit}(p, p_3, i)$ be the subjective, considered profitability $t_{ar} - t_{ao}$ attributed by p to principal p_3 after update i, or initially if $i = 0$, for $0 \leq i \leq n$ and $p_3 \in \mathcal{P}$.

Let $\mathcal{P}_{clique} \subseteq \mathcal{P}$ be those principals whose trust values were changed by one of the trust updates, or were part of one of the update actions. Then let $(\Delta t_{er}, \Delta t_{ar}, \Delta t_{eo}, \Delta t_{ao}) = \sum_{i=1}^{n} \Delta t_i$. Here, Δt_{ar} represents the total resource value received by p, while Δt_{ao} represents p's total costs. Equation 4.14 ensures that trust is conserved, so

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{p_3\in\mathcal{P}}(\text{profit}(p,p_3,i)-\text{profit}(p,p_3,i{-}1)) &= \Delta t_{ar}-\Delta t_{ao}\\
\iff \sum_{p_3\in\mathcal{P}}(\text{profit}(p,p_3,n)-\text{profit}(p,p_3,0)) &= \Delta t_{ar}-\Delta t_{ao}\\
\iff \Delta t_{ao} = \Delta t_{ar} + \sum_{p_3\in\mathcal{P}}(\text{profit}(p,p_3,0)&-\text{profit}(p,p_3,n))
\end{aligned}$$

Also, Equation 4.15 ensures that, $\forall p_3\in\mathcal{P}$, $\forall i: 0\leq i\leq n$,

$$\begin{aligned}
&\text{profit}(p,p_3,i)\geq 0\\
\vee\ &(\text{profit}(p,p_3,0)\leq 0 \ \wedge\ \text{profit}(p,p_3,i)\geq \text{profit}(p,p_3,0))
\end{aligned}$$

In particular,

$$\begin{array}{ll}
\text{profit}(p,p_3,0)\geq 0 \Rightarrow & \text{profit}(p,p_3,n)\geq 0\\
\text{otherwise,} & \text{profit}(p,p_3,n){-}\text{profit}(p,p_3,0)\geq 0
\end{array}$$

$$\begin{aligned}
\text{Hence, } \Delta t_{ao} \leq\ & \Delta t_{ar} + \sum_{p_3\in\mathcal{P}}(\text{profit}(p,p_3,0)-\text{profit}(p,p_3,n))\\
&+\sum_{p_3\in\mathcal{P}}\begin{pmatrix}\text{profit}(p,p_3,n) & \text{if profit}(p{,}p_3{,}0)\geq 0\\ \text{profit}(p,p_3,n){-}\text{profit}(p,p_3,0) & \text{otherwise}\end{pmatrix}\\
=\ & \Delta t_{ar}+\sum_{p_3\in\mathcal{P}}\begin{pmatrix}\text{profit}(p,p_3,0) & \text{if profit}(p,p_3,0)\geq 0\\ 0 & \text{otherwise}\end{pmatrix}
\end{aligned}$$

As the only principals whose trust values changed are the members of $\mathcal{P}_{clique}$, the other principals do not affect the sum above. Furthermore, Δt_{ao} represents the resource value received by $\mathcal{P}_{clique}$, and Δt_{ar} shows the resources they input, giving the desired inequality,

$$\Delta t_{ao}\leq \Delta t_{ar}+\sum_{p_3\in\mathcal{P}_{clique}}\begin{pmatrix}\text{profit}(p,p_3,0) & \text{if profit}(p,p_3,0)\geq 0\\ 0 & \text{otherwise}\end{pmatrix}$$

□

Lemma 4.5. *An attacker cannot systematically pump the system for useful, contracted resources.*

Proof: If an attacker could pump the system for resources, then eventually they would obtain more resources than they had contributed. This would contradict

Theorems 4.3 and 4.4, disproving the initial assumption that pumping was possible. □

These proofs show that the trust framework presented here provides protection against attacks where the attacker undertakes a large number of small, successful transactions, with a view to then cheating on a much larger transaction. This attack is typified by fraudsters on the eBay online auction site, who build up an excellent reputation selling low value items, then try to auction a bogus Ferrari to a gullible bidder [98].

The trust framework does still allow risk taking behaviour. However, continuous trust monitoring ensures that the risks are limited to the granularity of the accounting processes, and the resource outlay is limited to the perceived profitability of the recipient. Thus, by artificially inflating this profitability, larger risks would be allowed; however, this would not contradict the results above, since the trust model would still subjectively believe that it was in the black.

4.3 Trust Model for Implementing a Compute Server

This section outlines one of the applications of the contract framework and trust model. The goal of the compute server is straight-forward: to sell access to its computational resources in order to make a profit. While later chapters illustrate the generality of the framework in a wide range of applications, this section focuses on how the formal definitions translate into a practical application.

A compute server brings together the essential aspects of the contract framework: contract specification and resource accounting, trust management and contract selection.

4.3.1 Contract Specification and Resource Accounting

Each compute server contract specifies a piece of computer code that a server is to execute for the client, together with the resource requirements and terms of payment. Here, resources are identified by basic type, subtype and place, as shown in Table 4.2, measuring CPU time, storage, network bandwidth and currency. When a contract specifies its resource requirements, it may specify the subtype or place or both, or leave them empty if there are no specific

Basic Type	Subtype	Place	Units
CPU time	architecture (e.g. `ix86`, SPARC)	machine name	Speed in normalised CPU seconds
Storage	memory or disk	machine name	Kilobyte seconds
Network bandwidth	N/A	pair of machines, or default for flat rate	Kilobytes transferred
Money	currency	N/A	Units of currency

Table 4.2. Table of compute server resource types

requirements. For example, if the contract code to be executed requires a specific machine architecture, the contract may require resources of basic type CPU time and subtype `ix86`.

Therefore, in the formalism of Section 3.2, this means that the space of resource types $\mathcal{R}$ consists of triples of basic type, subtype and place.

Each quantum of resource usage is expressed as a resource type triple, a quantity of resources used, and a start and end time. (When representing these approximations in continuous terms, it is assumed that the resources were used evenly over the time interval specified.) The resources used by a contract are then represented as a collection of these `ResourceAtoms`. Formally, patterns of resource usage $\mathcal{R}_U$ are represented by `ResourceAtom` collections.

This model allows the compute server implementation to represent resource usage arbitrarily accurately if necessary, but also allows detailed resource usage patterns to be compressed into a more compact summary form too. The actual level of detail is thus decided on by the compute server: the more detail, the more accurate the resource accounting will be, but at the cost of higher overheads. Ultimately, the server aims to keep its overheads as low as possible, while still computing resource usage accurately enough to convince the client of the validity of payment request calculations and prevent the client from manipulating the measurements.

Resource usage is defined in compute server contracts to consist of two components: static and continuous requirements:

Static requirements. These express the needs of batch calculations, and are represented as the total resource requirements for each resource type, together with a maximum time within which they must be made available. They can also be used to represent the initialisation and startup requirements of multimedia applications. For example, a requirement for 10 normalised units of CPU time on any PC, within the first 30 seconds of the contract, would be expressed as `(ResourceAtom( cpuTime,ix86,'',10,None,None), 30)`. (The '`None`'s signify that the CPU resources are not otherwise limited to a particular time.)

Continuous requirements. These allow applications to ensure a constant supply of resources at regular intervals, such as that needed by multimedia applications. For example, to ensure 1 normalised unit of CPU time and 2 kilobytes of network bandwidth, per tenth of a second for 60 seconds, the contract would request `(ResourceAtom(cpuTime,ix86,'',1,None,None)+ ResourceAtom(network,'','',2,None,None), 0.1, 60)`.

If (v_1, t_1) represents the static resource requirements v_1 over a time period t_1 for contract c, and (v_2, t_2, t_3) represents the continuous requirements v_2 per t_2 seconds for a total of t_3 seconds, then this can be re-expressed formally, by analogy to Section 3.2.1, as

$$\mathsf{require}(c, t) = \sum_i \mathsf{require}_i(c, t)$$

$$\text{with } u \in \mathsf{require}_0(c, t) \iff u(t_1) - u(0) \geq v_1 \text{ if } t \geq t_1$$

and $\forall k \in \mathbb{N} : 1 \leq k.t_2 \leq t_3$,

$$u \in \mathsf{require}_k(c, t) \iff u(k.t_2) - u((k-1).t_2) \geq v_2 \text{ if } t \geq k.t_2$$

The compute server implementation 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 [101], 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 function 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 accounting function to specify a list of resource rate inputs, and initial values for any persistent variables. Whenever the accounting function 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 accounting function 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 accounting function 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.

An extract of the accounting language's BNF grammar is given in Table 4.3. The remaining rules are simplifications of the rules of the standard Python grammar [101], to allow only variable assignment, numerical arithmetic, and `if` ... `elif` ... `else` statements. There are no general rules for exception handling,

```
accounting_input: 'class' NAME '(' BASECLASS ')' ':' NEWLINE
    INDENT import_list var_list+ accounting_fn DEDENT
import_list: 'importList' '=' '[' [string_list] ']' NEWLINE
var_list: NAME '=' NUMBER NEWLINE
accounting_fn: 'def' PROCESSFN '(' '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 ']'
```

Table 4.3. Extracts from BNF grammar for compute server accounting language

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.

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 (such as `quantity` or `startTime`).

This accounting model has important formal and practical advantages. Formally, it provides an expressive mechanism for translating resource usage patterns $\mathcal{R}_U$ into other resources $\mathcal{R}_U$ expected in payment, and corresponds to the function $\mathsf{account}(c)$. Practically, the mechanism is well suited to resource constrained computation, because of the special properties of the accounting language: constant-time execution and incremental computation.

The accounting language ensures that each accounting iteration requires a constant, bounded quantity of resources. Because there are 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 `ResourceAtom` by the accounting function can be predicted. Thus the accounting overheads of a contract action are limited by the number of resource atoms generated. 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.

Incremental computation further ensures low marginal costs in accounting for additional resources consumed under a contract: accounting functions are expressed in terms of change in payment versus change in resource usage. This allows complex, stateful accounting functions to be specified while maintaining a fixed marginal accounting cost.

The textual representation of a contract illustrated below shows how these aspects are brought together practically, incorporating all the essential components defined in Section 3.2, apart from the client and server identities which are defined by the context:

```
contract.ContractTemplate( description   = "makeCPUload contract",
  resourcesRequired =
           ((ResourceAtom(cpuTime,ix86,’’,10,None,None), 30),
            (ResourceAtom(cpuTime,ix86,’’,1,None,None) +
             ResourceAtom(network,’’,’’,2,None,None), 0.1, 60)),
           # Both static and continuous resource requirements
  accountingFunction =
           """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)]
           """,
  codeToRun =
        """def runMe():
             import test.contracts.PBIL
             return test.contracts.PBIL.PBILServer\
                      (dest="taff:8085",10,6, events=events)
        """)
```

In this example, the code to be run by the server is defined to be a Python function, but this could equally be a reference to a Java method signature, or code in another language. To actually establish the contract, this contract representation would be incorporated in a signed contract message (as detailed in Section 3.3), sent between server and client. For communication efficiency in the compute server implementation, contract representations and also their accounting functions are cached by both parties (and identified by SHA-1 hash), so that each contract need be sent in full only once.

The example above is discussed in more detail in Chapter 7, as part of the implementation of the compute server and tests of its performance. The rest of this section defines the trust model for that implementation, and shows how it relates to the formalisms of this chapter.

4.3.2 Trust Management and Contract Selection

The compute server needs support from a trust model, in order to rationally select contracts which are worth its while to perform. In broad terms, this trust model follows the same form as the general purpose model defined in Section 4.2.1. However, that model does not specify the details of how the effect of recommendations is computed, nor the cost model to be employed; this section defines these for the compute server implementation.

Recommendations can be used both as a trust management tool, and to bootstrap servers and clients into a network. For compute server contracts, trust values $\mathcal{T}$ consist of 4-tuples of values of the form $(t_{er}, t_{ar}, t_{eo}, t_{ao})$. A recommendation then takes the following form:

```
Recommendation( Recommender = "http://server1:8011",
                Recommended = "http://server2:8015",
                Context = "",
                Degree = (5.0, 6.2, 5.6, 5.4),
                Limits = (2.0, 0.25),
                Expiry = "22 Oct 2003 11:30:00 +0000")
```

This recommendation is then encapsulated in a signed message, which authenticates its contents and allows it to be used as a certificate of trustworthiness. In the implementation, principals are identified by a URL, which is also used as their point of contact for establishing contracts. Each principal has a public and private key pair; recommendations and contract messages are signed with the private key, and the public key can be obtained either directly from the principal or from a trusted third party directory service, in the form of a well-known secure web site that maps URLs to public keys.

The example above shows a recommendation from principal p_1 (`"http://server1:8011"`) about p_2 (`"http://server2:8015"`), recommending that for all contracts (in any context), p_2 should receive extra trust $(5.0, 6.2, 5.6, 5.4)$, limited to a maximum of £2 or 25% of their existing trust, whichever is less. The recommendation is valid until 22 October 2003 at 11:30am, if appropriately signed.

Recommendations are used in the compute server as certificates of trustworthiness, to allow participants to stand surety for others. They are thus presented by clients to servers, and vice versa, during contract negotiations. Because of this constrained mode of operation, the implementation requires that the recommendations presented are free of cycles.

The integration of recommendations into trust values is best illustrated with an example: assume that participant p_2 is attempting to enter into a contract with p, with the help of the recommendation from p_1. Assume that p has an initial trust model T_m of its own direct experiences. If

$$\begin{aligned} T_m(p, p_1) &= (60, 50, 30, 40) \\ \text{and } T_m(p, p_2) &= (10, 10, 9.5, 9.5) \end{aligned}$$

then p sees the recommender p_1's net profitability as £10. As this is positive, the recommendation is worth considering, and p begins by discounting the recommendation according to p_1's behaviour in past contracts. (This is appropriate, since this is how any other contract originating from p_1 would be adjusted.) The degree of strength of the normalised recommendation is then

$$\left(\tfrac{60}{50} \times 5.0, \tfrac{50}{50} \times 6.2, \tfrac{30}{40} \times 5.6, \tfrac{40}{40} \times 5.4\right) = (6.0, 6.2, 4.2, 5.4)$$

This adjusts the expected return ratios to compensate for historical bias, and would also ensure the correct interpretation if principals were to recommend themselves with full trust. Secondly, the recommendation is scaled if necessary, to ensure that it does not exceed the limits of £2 or 25% of p_1's net profitability (£2.50), in terms of its contribution to either p_2's actual or expected profitability. In the example, there is no change as $|6.2 - 5.4| = |0.8| < 2$ and $|6.0 - 4.2| = |1.8| < 2$.

The new, normalised and scaled recommendation is then added to p_2's existing trust to obtain a considered trust value, so

$$\begin{aligned} T_{m,considered}(p, p_2) &= T_m(p, p_2) + (6.0, 6.2, 4.2, 5.4) \\ &= (16.0, 16.2, 13.7, 14.9) \end{aligned}$$

When there are many recommendations, they are all added iteratively to the trust model to obtain the final trust assignment. If some recommendations affect principals who themselves make other recommendations, then these recommendations are applied before their dependencies. Because recommendation cycles are disallowed, this process will eventually cover all of the recommendations presented. In formal terms, this resulting algorithm defines the function $\mathsf{consideredTrust}(T_m, Rec, c_1)$ of Equation 4.12.[5]

When recommendations are used for bootstrapping trust interactions, then each contract between client and server is potentially affected by two sets of recommendations: those already held for bootstrapping, and those presented under the contract. In that case, consideredTrust applies the local recommendations first, before applying those presented.

Trust values must also be updated as contracts progress, while taking recommendations into account. In the compute server implementation, this is achieved by apportioning responsibility for a principals trustworthiness between that principal and their recommenders, in proportion to their contribution to the overall apparent trustworthiness. For example, in the scenario above, p_1's recommendation contributed £0.80 of net apparent profitability to p_2's existing £0.50, giving a 62%:38% split in trust assignment. Thus a trust update of $\Delta t = (0.39, 0.39, 0.26, 0.26)$ would change p_1 and p_2's basic levels of trust to

[5] To ensure balance and maintain the properties defined in Section 4.2.1, principals that make recommendations lose the same amount of trust that they give to others, for the purposes of the contract concerned. This local loss of trust does not directly affect the other contracts they enter into, but does help protect against principals issuing numerous indirect recommendations to artificially boost the trust of a single principal. Chapter 7 discusses more fully other mechanisms for limiting the number of contracts principals enter simultaneously.

$$\begin{aligned}
T'_m(p, p_1) &= (60, 50, 30, 40) + \frac{5}{13} \times (0.39, 0.39, 0.26, 0.26) \\
&= (60.15, 50.15, 30.1, 40.1) \\
\text{and } T'_m(p, p_2) &= (10, 10, 9.5, 9.5) + \frac{8}{13} \times (0.39, 0.39, 0.26, 0.26) \\
&= (10.24, 10.24, 9.66, 9.66)
\end{aligned}$$

There are also a few special cases that need to be considered. To ensure that principals themselves can gain some profit, even if they start out with no or very little trust and depend on recommendations, a minimum personal contribution is defined, set to 5% in the current implementation. Thus, even if p_1 had contributed all of p_2's trust, p_2 would still gain 5% of the profits. The same would apply if p_2 had negative initial profitability. These rules are all combined to create a function $\mathsf{updatedTrust}(T_m, Rec, p_1, c_1, \Delta t, p_2)$ for Equation 4.13 which defines how trust values are updated as contracts progress.

After the recommendations have been considered, contracts are then selected on the basis of their expected profitability. In order to do this, principals need a model for pricing resource value, and a mechanism for estimating this value based on a contract's resource requirements. In the compute server implementation, each compute server p has a resource pricing function cost_p in which the costs are those it pays for its resources and for equipment depreciation. Clients, on the other hand, are supplied with contracts and funds over time (analogously to stride scheduling [104] in operating systems), and attempt to choose the cheapest server they can afford for each contract; they obtain recent market rates from a pricing server, to compute the cost of the different resource types. Finally, the uncorrected resource outlay of a contract (v_{best} in the formal definition) is taken to be the stated resource requirements, consumed continuously over the intervals specified. For example, in the contract specified above, this would amount to $10\frac{1}{3}$ units of `ix86` CPU time per second over the first 30 seconds, and 10 units per second over the next 30 seconds, as well as 20 kilobytes per second of bandwidth for the first 60 seconds. Section 4.2.1 then specifies how these resources are adjusted to correct for trustworthiness, and compute the expected return on investment that results.

This section has shown how a compute server implementation complies with the formal resource and trust models for contracts presented earlier. This compliance ensures that the proofs of liveness and safety in Sections 4.2.2 and 4.2.3 also apply to it. Conversely, the implementation also validates the formal model, by demonstrating that it leads to useful and practical implementations.

Trust modelling is essential in a distributed contract framework, to monitor performance and protect against loss of resources. In this chapter, a formal trust model for contracts has been developed, which offers provable safety and liveness guarantees. The usefulness of this model is further illustrated by applying it to a compute server implementation. This implementation is discussed in more detail in Chapter 7, together with performance results and analyses of its other security properties. First however, Chapter 5 extends the notion of resources to include both access control privileges and also unconventional factors such as trust and the cost of the user's time.

Chapter 5

Non-Computational Resources

This chapter explores the rôle of resources in contracts, including non-computational resources such as the user's time, and shows how authentication credentials can explicitly control contract accounting policies. Finally, a PDA collaboration scenario illustrates the other extreme, in which trust is manually controlled and the trust model is pre-eminent.

5.1 The User's Time as a Resource

This section develops the idea that the user's time can be treated as a resource in the contract framework. Explicit costing of users' time promotes appropriate security in computer systems, and encourages programs to shield users from unnecessary interruption. Furthermore, integrating this feature into the contract framework allows program code signing to be used to restrict access to both conventional resources and to the user's attention.

The user's time is a scarce and valuable resource in computer systems [40, 86, 94]. These systems therefore need to protect the user from unnecessary inconvenience or interruption. This is particularly true in multitasking environments, in which many different programs might be vying for the user's attention, increasing the risk of distracting the user from their train of thought.

As a result, the user's time needs to be explicitly costed, and offset against the value of getting an answer, to decide whether a question is appropriate. For example if access to a web page costs a fraction of a penny, it may be better to automatically accept the charge, instead of asking the user to consider accepting or rejecting it. Similarly, if security measures are too cumbersome then the user might decide to circumvent them, such as by logging in with a colleague's password if they forget their own [90]. These decisions can only be made appropriately if they take into account the value users attribute to their own time, and offset this against the realistic benefits of obtaining the information.

A two-pronged approach is needed to ensure respect for the user's time. Firstly, programs need to be aware of the value of that time. This value would not necessarily be static, but could vary with the user's activity. For instance, if a user had just been interrupted to answer a question, then it would be less disruptive to ask a second question at the same time rather than five minutes later. Similarly, the cost of interrupting the user as they read their email might be less than if they were busy typing or in a meeting.

Secondly, this respect needs to be enforced. Here, the contract framework can be used, by treating the user's time as a contractable resource. Programs would agree in advance on how much they would interact with the user, and would then be held to this agreement unless they negotiated a change of conditions.

In one sense, scheduling access to the user's time is comparable to CPU scheduling by the operating system: no process is ordinarily starved of access to the CPU; instead, the rate of access is limited in favour of other tasks. The difference is that apparent user idleness (from the computer's perspective) is also an important task, unlike the scheduling of CPU idle time.

5.1.1 Self-Financing Web Services

A self-financing web server demonstrates how resource pricing can interact with the pricing of the user's time. In this example, a web server assigns prices to each of its web pages, and charges users who ask to view them. For example, news articles might be charged at a price of 2p each, while the main page and any images might be provided free of charge (to take advantage of web caches for the bulk of the bandwidth).

The contract framework could simply be applied to this application by entering into a new contract for every page downloaded. However, if most users were expected to access more than one page, then it would be more efficient to instead start a single contract upon the first request, covering all of the pages on the site, followed by payment for page views at set intervals. This would be particularly appropriate for resources of low value, for which even existing

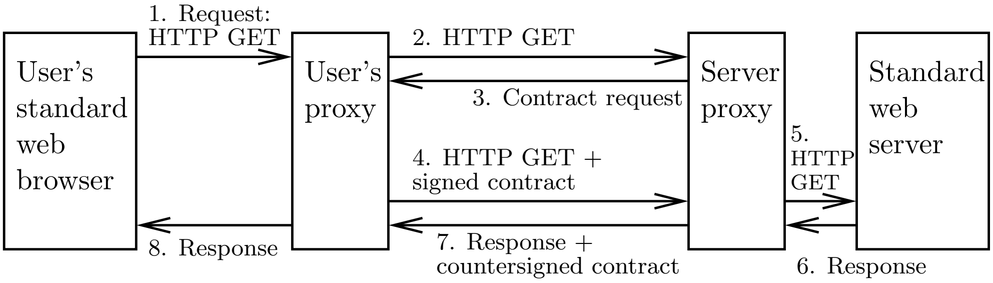


Figure 5.1. Pricing information on the Internet

micro-payment systems are not cost-effective because of excessive overheads — the use of contracts would allow fewer, larger payments for these services.

The web server's resource model treats page requests as its resources, instead of conventional resources such as CPU time; the subtype of a resource denotes its URL and hence its pricing. Thus page requests can be priced using resource accounting functions, as part of the contracts between client and server. Furthermore, since all of these requests fall under the same contract, sophisticated pricing policies can easily be incorporated, such as progressive volume discounts for multiple requests.[1]

This addition of contracts to the Internet is made nearly transparent, by inserting contract-aware proxy servers between standard web browsers and web servers, as shown in Figure 5.1. On the browser side, contract information is added to HTTP requests, while on the server side this information is used to charge users for the information they requested. Simultaneously, the user's proxy monitors its costs and contract information, occasionally interrupting the user for extra information such as to confirm contracts.

A naïve implementation of the user's proxy server in Figure 5.1 would ask the user to confirm every contract message and payment explicitly, to ensure that they knew of all the costs and charges. A more sophisticated implementation can better support the user however, by acting as an intelligent intermediary:

- The proxy is given a discretionary resource allowance, which it can use to make payments automatically.[2]
- Beyond this budget, the user is contacted to validate or reject payments, either once or repeatedly for that contract, e.g. 'accept this payment seven times over'.

[1] With iterated single requests, this would still have been possible, but far less transparent to the user and more costly to implement.

[2] This allowance is either paid continuously over time, or earned by the proxy as a fixed mark-up on resource prices.

- The user's proxy enters into *explicit* contracts with the server's proxy. But there is also an implicit contract between the user's proxy and the user, in which the user's time is one of the resources.

The contract between the user and their proxy provides the proxy with funds, in exchange for data from the self-financing web servers and access to the user's time. In this scenario, there is no need for trust management or formal contract exchanges between the user and the proxy; the contract framework is instead used at the design stage to engineer the proxy's actions. The proxy server tries to optimise its subjective profitability by choosing between automating decisions and asking the user for guidance — but asking for guidance has a cost, as set by the user. The proxy is also constrained in its actions to limit its autonomy:

- The proxy may not automatically enter new contracts, but may extend existing ones.
- The proxy may not overspend its discretionary financial budget.
- Decisions authorised by the user are funded from the user's purse. If they support the proxy's earlier automatic decisions, the proxy is refunded its discretionary spending on them.

Finally, in situations where users pay for the bandwidth which they use, such as when using a GPRS mobile telephone for Internet access, the user's proxy can assess the bandwidth costs too when retrieving web pages and other information. For example, only the first portion of a very long document might be downloaded, until the user had decided whether or not it was relevant.

Thus treating time as one resource among many enables computer programs to rationally avoid distracting the user with unnecessary questions, without compromising safety for important decisions.

5.1.2 Code Signing and Resource-Limited Sandboxes

The user's time can also feature explicitly in contracts. This allows code signing and sandboxed operation [44] to be extended to allow resource limits and protection of the user from interruptions.

Code signing conventionally limits which resources a program has access to; for example, a Java applet ordinarily has no access to the local disk storage, or to sites on the Internet other than its host web server. However, the program code can be signed by a trusted supplier to authorise its access to these resources. As with other access control systems, this access is normally granted on an all-or-nothing basis for a particular group of resources. For example, an applet might be given read/write access to file `/tmp/test.log`, but it could then make arbitrary use of that file, filling the drive with data or constantly changing the file's contents.

Even an untrusted applet can conventionally cause a local Denial of Service attack, by monopolising the CPU and available memory, or generating too many threads for the operating system scheduler to manage. Although these difficulties can be overcome by setting extra operating system limits for the Java virtual machine, these same limits would then have to apply to all applets, or be manually configured for each. The essential difficulty is that access control grants rights on a per-resource basis, not on the basis of *usage* — there is zero marginal cost to the applet for actually using the resources, even for scarce shared resources.[3]

Instead, code signing can be extended using the contract framework of this thesis, allowing applets or programs to express their resource needs as an attached contract. This allows a resource-limited sandbox to be created, free from the risks of Denial of Service attack associated with conventional sandboxes — provided that the signed contracts encompass all resources in the system. (Section 5.2 discusses in more detail the trade-offs between direct and indirect resource representation.) Furthermore, since each signed applet would automatically receive a resource allocation from the user (by default, enough for the resources it requested), the user could simply adjust this allocation linearly to promote or restrain a task.

This algorithm reduces directly to CPU stride scheduling [104] in the simple case where CPU time is the only resource considered; such proportional-share algorithms typically show better responsiveness for multimedia applications than traditional priority-based algorithms (augmented to promote idle tasks) [87].

Code signing with contracts still supports the access control restrictions of ordinary code signing, but also allows more specific resource usage limits to be added. In this context, the contract is between the code signer (represented by the code they produce) and the user's computer; the resource requirements are specified in the signed code, and the accounting function is specified by the user

[3] The applet could try to use proxy measures, such as changes to the delay in completing requests, to try to detect resource contention, but at the risk of giving up resources only to have them taken by less cooperative contenders.

to represent the availability of resources. Since the user decides on the resources to give to the signed code to adjust its responsiveness, trustworthiness is not an essential metric, although it can be used to report to the user which programs are trying to exceed their resource limits.

These contracts can include the user's time as a resource too: interrupting the user carries a cost for the signed code. Here, interruption is defined as opening a new input window when another task has the user's mouse focus. Because signed code is aware of its contractual terms, it can limit its attempts to contact the user appropriately; if it does not, its input requests may be delayed until it has enough resources available.

For example, a user might run an MP3 player, which would request enough CPU resources and bandwidth to play its data streams in real time. If it was later given a very high bitrate MP3 stream to play, it might try to renegotiate its contract to ensure enough resources. Failing this — if the user declined the request or the communication budget with the user was exhausted — it would have to hope for more resources than promised, or play the track with lower fidelity, or pause or skip it entirely. Similarly, a peer-to-peer file sharing service might be installed by a user under contract to upload data at no more than 3KB/second to avoid saturating the user's network link. Here too, the service would agree to limit its interruptions of the user (e.g. upgrade notification messages), while the user could tweak this by configuring the value of their time.

Thus contract resources need not be only computational or financial; this section has shown how the user's time can be integrated into a resource framework, as can operational resources such as web page accesses and file operations. The following section extends this latter idea further, by showing how not just actions but also access control credentials can be integrated into the resource framework.

5.2 Resources and Credentials

Resources hold the information against which contracts are assessed. This information need not just come from performing the contract, but could also incorporate outside information too. In this section, access control credentials and trust information are examined as contract resources, demonstrating the duality of trust and resources.

5.2.1 Access Control Credentials as Resources

Imagine a contract such as ‘Web pages cost 2p each to download, or 0.1p each for subscribers’. In the contract framework examples presented thus far, this would have to be organised as two separate contracts, with an initial contract negotiation phase to decide which version would be offered.

Instead, a special resource type can be created to identify subscribers: a subscriber that holds an `IsSubscriber` rôle membership certificate can present this to the server as proof. The server would then give the subscriber’s contract a `ProvenSubscriber` resource item, which the contract accounting function would use to adjust its pricing function.

Unlike conventional resources, these `ProvenSubscriber` resources would not be conserved, but would be created whenever needed by a server. Thus the resource does not represent the credential itself, but rather the property of having the credential. These access control credential resources have a number of advantages, but with corresponding limitations too:

- Adding credentials to contracts reduces the need for extra contract negotiation, as credentials need not be exchanged before the terms are agreed, but the resulting contracts are longer, and harder to analyse automatically.

- Credential augmented contracts provide their participants with extra information, which they can use to better optimise their costs, but this information may be confidential, in which case early credential exchange would still be needed.

- Contract terms can change on the fly when a new credential is received, but this reduces the predictability of contract value.

Thus access control credentials can sometimes still be needed in initial contract negotiation, but can also greatly improve contract flexibility if they are treated as contract resources albeit at some extra cost.

The example above can be represented in the formalisms of the resource framework, with a space of resources $\mathcal{R} = \{\mathcal{R}_{\text{money}}, \mathcal{R}_{\text{web}}, \mathcal{R}_{\texttt{ProvenSubscriber}}\}$ where $\mathcal{R}_{\text{money}}$ is money in pounds, $\mathcal{R}_{\text{web}}$ represents a web page request and $\mathcal{R}_{\texttt{ProvenSubscriber}}$ is the pseudo-resource identifying subscribers. Then the accounting function described at the beginning of this section is defined by:

$$\forall u \in \mathcal{R}_U, \forall t \in \mathbb{R}, \forall R \subseteq \mathcal{R},$$
$$\mathsf{account}(c)(u)(t,R) = \begin{cases} k \,.\, u(t, \{\mathcal{R}_{\text{web}}\}) & \text{if } \mathcal{R}_{\text{money}} \in R \\ 0 & \text{otherwise} \end{cases}$$
$$\text{where } k = \begin{cases} 0.001 & \text{if } u(t, \{\mathcal{R}_{\texttt{ProvenSubscriber}}\}) > 0 \\ 0.02 & \text{otherwise} \end{cases}$$

This function charges money for the number of accesses to the $\mathcal{R}_{\text{web}}$ resources, and scales the charge calculated at time t appropriately according to whether any $\mathcal{R}_{\texttt{ProvenSubscriber}}$ resources had been received by the contract by that time. In this function, a new subscriber is effectively refunded the difference in price for pages viewed earlier under the same contract before subscribing; the function could equally have been defined not to refund this. Parameterized credentials [7] can also be incorporated in the model in the same way, by appropriately subtyping the new resources.

Incorporating access control credentials directly into the contract framework does not negate the need for conventional access control systems. These dedicated systems allow sophisticated policies to be expressed, analysed and managed on a large scale, with special mechanisms for credential revocation and negotiation. However, the example above shows how rôle-based access control credentials can feature explicitly in a resource contract, allowing better integration of contracts with existing access control systems.

The subscription example also shows how actions such as retrieving a web page can be treated as resources too. This approach can interact with credential resources, to allow access control credentials to effect resource limits via the resource pricing system. For example, holders of rôle credential `IsGuest` might be entitled to view five web pages for free as a promotional trial, but no more. This can be achieved by giving the first 5 accesses a cost of zero, then setting the cost impossibly high for subsequent access. Rather than choose an arbitrary price for this, we introduce a new resource type $\mathcal{R}_{\text{impossible}}$ which nobody can ever obtain, to use in enforcing the policy. This is illustrated in the following accounting function, which follows the same template as the compute server policies of Section 4.3:

```
class Accountant(resources.ResourceContract):
   IsGuest = 0
   FreeWebPages = 0
   IsSubscriber = 0
   def processResourceAtom(self, atom, imports):
      if atom.type == resources.provenGuest:
       self.IsGuest = 1; return []
    if atom.type == resources.provenSubscriber:
       self.IsSubscriber = 1; return []
    if atom.type == resources.web:
      if self.IsSubscriber:
        return [ResourceAtom(resources.money, '£', '',
                             0.001*atom.quantity,
                             atom.startTime, atom.endTime)]
      if self.IsGuest:
        self.FreeWebPages += atom.quantity
        if self.FreeWebPages <= 5:
          return [] # Free trial
      # Prohibit web page views for users who have exceeded the
      # free trial limits, or are neither guests nor subscribers.
      return [ResourceAtom(resources.impossible, '', '',
                           1, atom.startTime, atom.endTime)]
    return [] # Other resource types are free.
```

If the web server checks and updates its accounts before serving each page, then no guest contract will be allowed more than 5 pages as they will be unable to provide the necessary $\mathcal{R}_{\text{impossible}}$ resources before retrieving the page. Otherwise, the guest account will be suspended automatically if it exceeds its limit, at the next accounting iteration.[4] (For those automated applications in which a manual override is needed, such as health information systems, $\mathcal{R}_{\text{impossible}}$ could be replaced in the accounting function with another resource type available only through the override process.)

Thus prospective enforcement of accounting policy allows access control enforcement from within the contract framework. Although this would be more expensive to implement in this way than with a dedicated access control solution, the advantage is that it allows complex limitations to be expressed in terms of resource usage combinations, which would not be possible with conventional access control systems.

Using novel resource types also highlights the trade-off between resource model completeness and complexity. On the one hand, the resource model aims to protect a system from attack or loss by ensuring that all system resources are accounted for, suggesting a minimal, low-level resource model that directly represents resources such as CPU time and network bandwidth. On the other hand, users of a service expect pricing in terms of the services provided, not the

[4] In this accounting function example, guests can become subscribers to continue to pay for and receive web pages, but non-subscribers cannot retrieve anything without becoming a guest.

mechanisms used to offer them. The risk is that this can lead to incomplete, indirect models that are unable to measure or detect attacks, and therefore unable to protect against them.

There are two ways to protect indirect resource models from these attacks: complete interface instrumentation, or covert modelling of extra resources in contracts.

Complete instrumentation assumes that a system's direct resource usage can always be attributed to some part of its interface to the outside world, and that each interaction leads to a bounded amount of resource usage. Therefore, monitoring all aspects of this interface allows bounds to be placed on the total direct usage, and so with restrictions on these interfaces, the total resource usage can be controlled. The disadvantage is that the monitoring and control is very indirect, and based on maximum resource usage bounds as opposed to actual usage. Thus the effects of some actions may be overrated, causing them to be curtailed unnecessarily. Nevertheless, provided all resources have non-zero cost, resource losses can always be identified.

Covert modelling extends contracts as they are being established by adding an extra layer of accounting for the direct system resources; this supplements the accounting contract agreed between client and server. This extra layer can either be independent, or be created by rewriting and extending the existing accounting functions. The extra modelling ensures that all resources are taken into account internally, and can limit access to them when this is unexpectedly high. This approach is similar to that taken by many online email service providers to detect spammers using their services: only mailbox size and message length are nominally limited, but the email service also tracks each user's bandwidth and the number of messages which they send. Accounts which exceed limits on these secondary metrics are then suspended as suspected spammers.

Covert modelling allows abnormal resource usage to be detected and controlled directly. This does introduce the risk that the service may then be seen as unreliable or unpredictable by those users affected, resulting in a loss of trust, since they continue to assess the contract in terms of the agreed metrics only. Thus covert modelling needs to be used carefully, with restrictions applied only very occasionally and preferably to slow down access rather than disallow it completely.

Access control systems can be integrated well with the contract framework; credentials can be represented as contract resources to express sophisticated resource pricing policies, and when the controlled actions are also exposed as resources, these policies can help enforce access restrictions. Lastly, these resources can help protect a system from outside attack, but only as long as all resource usage is monitored, either indirectly at the interfaces or directly from within.

5.2.2 Trust as a Resource

Trust values may themselves be seen as resources in the contract framework, albeit associated with principals instead of contracts — and in contrast with the conventional view of trust as part of the contract control structure. This contrast helps illustrate the contention between information and control in the contract framework, and shows why this distinction is simultaneously both important and irrelevant.

Since trust values are a reflection of the recipient's contract actions over time, these could be represented as resources within the contract resource model. For example, in the contract trust model of Section 4.2.1, a trust value is a four-tuple of the form $(t_{er}, t_{ar}, t_{eo}, t_{ao})$. Each of these components could then be represented as a resource type, and updated as the contract progressed. In one sense, this would reduce the independence of the contract resource model from outside information, although it could be argued that all resource usage is subject to outside influences. Besides, simulation of contracts for resource price estimation purposes relies on the fact that accounting functions do not need to communicate with the outside world to generate accurate results, not that they do not affect it.

Recommendations could then be seen as credentials authorising the transfer or trust resources between principals, much as access control credentials authorise the use of other resources. Recommendations could even act as resources themselves, by analogy with the previous section, and be incorporated explicitly into the resource pricing of contracts.

Furthermore, certificates attesting to past successful contracts could be used by participants to obtain better contract terms, either as resources themselves or as a source of extra trust.

What would this add to the contract framework? The separate resource and trust models would be replaced with a single unified model, and a unified accounting function would factor in both direct costs and trust-based discounting in order to simultaneously price resource usage and update its resource tallies. The result would be a less restrictive but weaker model for contracts than that presented in Section 4.2.1. Thus this new model would be at least as expressive as that 'general purpose contract model' — the existing model is simply a specific instance of it — and it can also be used to generate new families of contract frameworks.

However, this new unified model has less intrinsic structure than its predecessor, making the management of contracts more open-ended and harder to analyse. For example, instead of necessarily pricing resource usage and adjusting for trust in order to choose profitable contracts, contracts could be chosen on any basis at all. Similarly, the properties of contract liveness and safety proved in Sections 4.2.2 and 4.2.3 would no longer necessarily hold with these new open-ended 'contracts'.

In the existing contract model, there is a clear distinction between the information associated with a contract (the resource usage) and the control structures (accounting functions and trust values). The unified model presented above shows that this distinction is only a matter of perspective, constrained partly by the choice of contract model. At the same time, this distinction is necessary, because the structure it imposes confers useful, general properties on contracts, and makes their analysis tractable.

In summary, this section has shown the richness and expressiveness of the contract resource model, which can incorporate both credentials and the actions they govern. However, extending it still further and conflating trust and resources simply weakens the model and its properties, without offering clear advantages. For a different perspective on the rôle of trust and resources, the following section defines a PDA collaboration scenario which uses only elementary resources but features a sophisticated model of trust transfer.

5.3 PDA Collaboration

The principles of the contract framework can also be used to create models of trust and resources that do not clearly follow the conventional contract model, but are more tightly defined than the 'unified model' of the previous section. For example, in a PDA address book scenario, users need strictly controlled trust management so that they can limit the spread of their personal information,

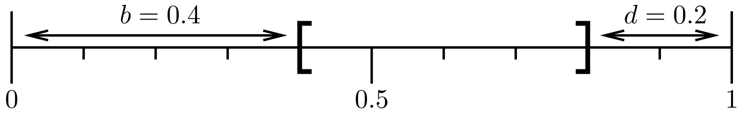


Figure 5.2. Visualising a belief-disbelief pair $(0.4, 0.2)$

while the scope for automatic monitoring of actions is considerably more limited than for fully computerised applications such as compute servers.[5]

In the address book scenario, the users structure their personal information by linking together related items; for example, Alice's telephone number, name and address would all be linked to an item representing her identity. Some of these items might be confidential and others public knowledge, e.g. Alice's personal mobile telephone number as opposed to her company switchboard number. This confidentiality of information is expressed by linking the entries to appropriate categories.

Each link is also labelled with the confidence the linker has in it, and signed to certify its authenticity. This confidence measure is represented by a pair of numbers (b, d) which signify respectively the strength of belief and/or disbelief in the link, with the constraint $b+d \leq 1$. Here, $(1, 0)$ represents certain knowledge, $(0, 1)$ is pure disbelief and $(0, 0)$ gives no information at all. This representation can be compared to Jøsang's logic of uncertain probabilities [52], which is in turn based on the Dempster-Shafer theory of evidence (see Section 2.2). However, the trust values here are chosen to be read directly by human users rather than formally grounded in statistics, since the trust structures are to be formed by the users themselves. In keeping with this intuitive approach, the strength or weight of a recommendation is given by $b - d$ and is used to help decide which links to accept.

Belief-disbelief pairs can also be represented graphically as intervals on a unit line, as illustrated in Figure 5.2. If the belief is stronger than the disbelief, then the midpoint of the interval will lie to the right of the 0.5 mark, showing how this view also gives the user practical assistance in understanding the data. (The midpoint lies at $x = b + \frac{1-b-d}{2} = \frac{1}{2} + \frac{b-d}{2}$. Thus $x > \frac{1}{2}$ iff $b > d$.)

Each labelled address book link is treated as a recommendation from the issuer, thus if Bob links category 'work' to Alice's phone number 763-621 with confidence $(0.4, 0.2)$ then this recommendation signifies that Bob trusts mem-

[5] This PDA scenario and its trust calculation functions extend earlier work with Nathan Dimmock and Jean Bacon, published in a paper at the PerCom 2003 conference [94]. That paper demonstrated that the PDA address book was an implementation of the SECURE EU project's trust and risk models.

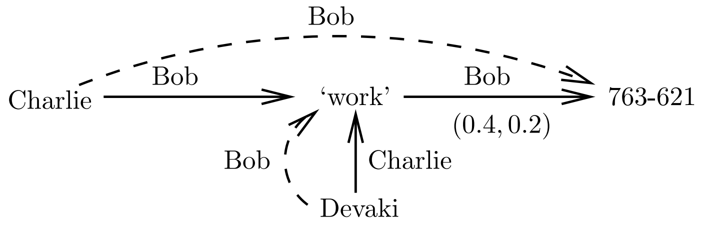


Figure 5.3. Dashed lines show how PDA recommendations are chained together

bers of that category to know Alice's phone numbers. Recommendations are also automatically chained together, thus if Bob also recommends that Charlie be allowed to read work contracts, then Bob will allow Charlie to read Alice's phone number.[6] Recommendations are also combined to allow delegation of privileges, so that if Charlie recommended Devaki for the work category, then Bob would honour that recommendation and trust her too, albeit not with more weight than Bob's trust of Charlie as a work colleague.

Thus recommendations essentially treat the friend of a friend as a friend, allowing social networks to be simulated. Figure 5.3 shows the recommendations described above, with the deduced recommendations marked with dashed lines. Each arc is labelled with the recommender's identity and the strength of the link (if specified in the text), and the arc destinations show the recommendation contexts. These recommendations have a similar structure to those described in Chapter 4, but they differ in that the context represents not a family of contracts but rather a subspace of the trust space.

These recommendations also act as permissions, e.g. 'Bob gives Charlie permission to read the work category', but in this section the primary concern is instead their representation as a trust model, and the resulting operational model.

Trust values in this application, assigned by principals to each other, map items to belief-disbelief pairs. In the notation of Section 4.1, this means that the space of trust values $\mathcal{T}$ is defined by

$$\mathcal{T} : \mathcal{P} \to T_b, \text{ with basic trust values } T_b = \{(b, d) \in [0, 1] : b + d \leq 1\}$$

and the trust model is then $T_m : \mathcal{P} \times \mathcal{P} \to \mathcal{T}$.

Here, the space of principals $\mathcal{P}$ has been extended to also include not just

[6] In fact, it would be Bob's PDA that would allow Charlie's PDA to read the number. Similarly, principal identities also refer to people's computational representatives and only indirectly to the people themselves. However, this distinction is often glossed over in this section when there is no ambiguity.

From To	$\mathcal{A}$	$\mathcal{C}$	$\mathcal{D}$	$\mathcal{P}_A$	$\mathcal{P}_L$
$\mathcal{A}$	✓	✓		✓	
$\mathcal{C}$				✓	
$\mathcal{D}$					✓
$\mathcal{P}_A$					
$\mathcal{P}_L$					

Table 5.1. Recommendation linkages allowed

'actors' or human principals (identified by their public cryptographic keys), but also the category permissions and data item identities. The complete space $\mathcal{P}$ consists of the following:

Actor Permissions $\mathcal{A}$ are used to identify people (and their aliases), such as 'Alice' or 'asa21'.

Category Permissions $\mathcal{C}$ represent membership of a category such as 'work'.

Data Entry Permissions $\mathcal{D}$ refer to address book entries, including telephone numbers and names in this application.

Action Permissions $\mathcal{P}_A = \{Read, Write\} \times \mathcal{C}$ allow the holder to read or write data in a particular category.

Link Permissions $\mathcal{P}_L = \{Link\} \times (\mathcal{A} \cup \mathcal{C})$ are used when data is written, to recommend that it be associated with a category or an actor.

Only certain pairs of permissions may sensibly be linked by recommendations, as illustrated in Table 5.1. These links are then chained together to compute the resulting trust values for the trust model T_m.

5.3.1 Trust Assignments as a Least Fixed Point

When many recommendations affect the same item, they need to be combined sensibly to produce the resulting trust values. In the field of access control systems, this consistent interpretation can be expressed as the least fixed point of a system of equations [107]. While those decisions are binary, and PDA collaboration produces trust intervals, the same technique can still be applied here by defining proper orderings over the intervals. (The approach of using least fixed points for trust and recommendation systems has been developed by the SECURE project [16].)

Two possible orderings for T_b order values by trustworthiness or by information. The trustworthiness order $\preceq$ is defined by $(b_1, d_1) \preceq (b_2, d_2)$ iff $(b_1 \leq b_2)$ and $(d_2 \leq d_1)$, which forms a lattice on our trust domain T_b and has bottom element

$(0,1)$. However, it is the second natural ordering $\sqsubseteq$, according to information, where $(b_1, d_1) \sqsubseteq (b_2, d_2)$ iff $(b_1 \leq b_2)$ and $(d_1 \leq d_2)$ with bottom element $(0,0)$, which we will use in combining recommendations below.

Users must combine their own recommendations with others' to assess trust. This is achieved by forming a *policy function* for each principal's recommendations; these policy functions are then combined to reach the appropriate trust conclusions. Each policy function denotes the trust each principal places in others' trust information; $Pol_x(T, y, z)$ is the degree to which principal x believes y should hold permission z, if everyone else's trust assignments are given in T.

This combines x's own recommendations with recommendations by others x trusts. Let $d_x(y, z)$ summarise x's recommendations, with $d_x(y, z) = t$ if x recommends that y be linked to z with certainty t, and (0,0) otherwise. (Newer recommendations are assumed to supersede older ones.) Two sorts of recommendations are then transitively combined, generalising the chaining shown in Figure 5.3:

- those where x associates y with p, and p with z, and
- those where x gives p permission z, and p recommends y for z.

This is summed up in the policy function

$$\begin{aligned} Pol_x(T, y, z) = \bigoplus \Big\{ & \bigcup_{p \in \mathcal{P}} T(x, y, p) \otimes T(x, p, z) \cup \bigcup_{p \in \mathcal{P}} T(x, p, z) \otimes T(p, y, z) \\ & \cup\ d_x(y, z) \Big\} \end{aligned} \tag{5.1}$$

where

$$Pol_x \ : \ (\mathcal{P} \rightarrow (\mathcal{P} \rightarrow (\mathcal{P} \rightarrow T_b))) \rightarrow (\mathcal{P} \rightarrow (\mathcal{P} \rightarrow T_b)) \tag{5.2}$$

$$(b, d) \otimes (e, f) = \begin{cases} (0,0) & \text{if } b \leq d \\ (\frac{e}{k}, \frac{f}{k}) & \text{otherwise} \end{cases} \text{ with } k = \max\left(\tfrac{e}{b-d}, \tfrac{f}{b-d}, 1\right) \tag{5.3}$$

The $\otimes$ operator acts in the same way as the discounting operator in Jøsang's logic, to adjust trust values produced by another principal according to their perceived trustworthiness. The operator here ensures that only principals with positive net trust weight can make recommendations, and that these recommendations cannot be stronger in effect than the original weight of trust in the recommender.

We also define $\bigoplus X_i$ to combine a number of recommendations monotonically (with respect to $\sqsubseteq$), by averaging their belief and disbelief components re-

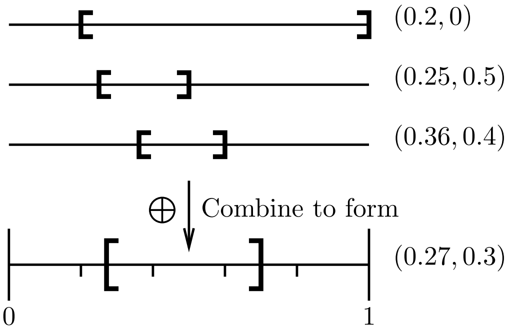


Figure 5.4. Combination of recommendation certainties

spectively. For example, given three recommendations $(0.2, 0)$, $(0.25, 0.5)$ and $(0.36, 0.4)$ as shown in Figure 5.4, the compound recommendation deduced by $\bigoplus$ is $(0.27, 0.3)$.

Informally, this considers the influence of the original recommendations d, together with the recommendation chains shown above. These are then combined to deduce an updated trust value. By repeating this process, the trust values converge to a final trust assessment.

To guarantee this convergence, each policy function Pol_x must be monotone with respect to T, as shown in equation 5.4. If we then combine all the individual policy functions into a single function $Pol(T)$, this will also then be monotone, and have a least fixed point $T_m = Pol(T_m)$, which will be the final trust model and considered trust assessment.

$$T' \geq T \Rightarrow Pol_x(T',y,z) \geq Pol_x(T,y,z),\ \forall x,y,z \in \mathcal{P} \quad (5.4)$$

$$Pol(T)(x, y, z) = Pol_x(T, y, z),\ \forall x,y,z \in \mathcal{P} \quad (5.5)$$

$$Pol : (\mathcal{P} \to (\mathcal{P} \to (\mathcal{P} \to T_b))) \to (\mathcal{P} \to (\mathcal{P} \to (\mathcal{P} \to T_b))) \quad (5.6)$$

The policy function Pol given above always converges for non-cyclical recommendation sets, such as those used in the address book application. However, it is not always monotone as it stands — it needs to be augmented to ensure monotonicity, and hence convergence under recommendation cycles.

Therefore, in computing the trust policy, we augment each trust value T_b with a list of 'parent' recommendations that contributed to it. As the trust policy calculation iterates, the parent lists increase in terms of an extended information order $\sqsubseteq'$, whose bottom element $\perp_{\sqsubseteq'} = (0,0, [])$ represents no recommendations at all. The least fixed point will then correspond to the least specific trust assignments justified by the available recommendations.

Extensions are also needed for the operators $\otimes$ and $\bigoplus$ to propagate parent lists to the deduced recommendations, and to ignore any cyclical contributions from recommendations whose parents include the recommendation currently being computed. This ensures theoretical monotonicity, while still producing the same deduced trust values as before in the absence of cycles.

Thus the least fixed point of the policy function Pol produces a considered trust model T_m, which consistently combines all recommendations in order to deduce the trust that should be placed in each link.

5.3.2 Trust for Decisions

Trust calculations ultimately lead to decisions about which actions a principal will allow to be performed. As with the contract framework, this is based on an economic analysis of the expected costs and benefits involved, and the decisions are then prioritised based on the expected utility of each.

This is done with the help of a hierarchy of information categories (such as 'work', 'friends', 'personal') which are used to assign an economic value to each piece of information — the user assigns a value val_c to each category c. These categories are also ordered by the user who arranges them in a lattice structure, e.g. allowing 'personal' acquaintances to read their friends' contact details. Finally, the user also configures two other values; val_{read} represents the importance of providing information to others, in terms of the value of the expected goodwill in return, and val_{time} defines the cost to the user of being interrupted. These two values would be reconfigured by the user depending on the context, such as whether she was busy in a meeting or trying to exchange phone numbers with colleagues.

Whenever an address book entry is accessed, a decision is needed on whether to provide the information automatically, query the PDA owner for confirmation, or refuse the request. However, the cost of the owner's time also needs to be taken into account in deciding whether to interrupt them — this extra cost affects the expected benefit of asking, and only if a net benefit is expected is the user interrupted, otherwise an automatic 'no' is given to the request for access.

Principals and data may be associated with multiple categories in an address book, for example some work colleagues might also be considered friends. Thus if Alice's PDA is considering allowing Bob access to data item d on her PDA, it must consider all pairs of categories (c_p, c_d) for which members of c_p are allowed to read from c_d with certainty $(b_1, d_1) = T_m(Alice, c_p, Read\, c_d)$ and $b_1 - d_1 > 0$, when deciding on the appropriate response to a request. In practice, only a few of these permutations are relevant for a particular request, so the extra overheads involved are small. Furthermore, this data can be calculated in advance before the request is made, as it depends only on Alice's category lattice.

Alice's PDA then calculates the appropriate costs and benefits on her behalf. Let (b_2, d_2) be Alice's considered trust in Bob's membership of c_p, to which Alice has assigned value val_{c_p}.

There is clearly a benefit (from Alice's perspective) to not giving out data in c_d if she does not believe that Bob should have access to it, signified by recommendations yielding $b_2 < d_2$. This benefit can be defined as

$$\text{Benefit}_{no}(b_2 - d_2, val_{c_p}) = -val_{c_p}.(b_2 - d_2)$$

To calculate Benefit_{yes}, Alice's PDA considers how strongly Bob is associated with c_p and the expected benefit of that association, set against the importance of the data he is trying to read, which encodes the potential cost of Bob ignoring Alice's recommendations and redistributing it indiscriminately. The function represents the benefit of helping someone who is well trusted to read low-value information, while requiring greater assurance to allow access to more valuable information.

$$\text{Benefit}_{yes}(b_2 - d_2, val_{c_p}, val_{c_d}) = val_{c_p}.(b_2 - d_2) - \max(val_{c_d} - val_{read}, 0)$$

This definition balances the expected benefit of assisting Bob, $val_{c_p}.(b_2 - d_2) + val_{read}$, against the value of the data being read val_{c_d}, while preventing the benefit of giving out information from exceeding the expected benefit of interacting with Bob, even if the value of val_{read} is greater than the value of the data involved.

The last cost function determines whether it would be worthwhile to interrupt Alice for a decision, based on her trust in Bob and the expected value of the information to be read, offset against the cost of her time.

$$\text{Benefit}_{ask}(b_2 - d_2, val_{c_p}) = val_{c_p}.(b_2 - d_2) - val_{time} + val_{read}$$

Alice's PDA performs these benefit calculations, prioritising them so that an automatic 'no' trumps a 'yes', which in turn overrides an 'ask' decision. If no decisions have a net benefit, then the decision is a 'no'.

5.3.3 Comparison with the Contract Model

This application is relevant to the contract framework because it presents a very different model for applying trust and the cost of user's time to support automatic decision making. While the PDA scenario can be moulded to fit into the existing contract framework, it is more appropriate first to question why it does not fit immediately, and decide whether this is a limitation of the scenario or of the framework.

The most important differences between this application and the others are:

Contract duration In the PDA scenario, each decision is made atomically and in isolation, based on the information available at the time. As a result, there is no need to track resource usage over time, unlike ordinary computational contracts which must be reassessed continually as they progress.

Automatic assessment Computational contracts are able to assess the outcomes of their decisions, and can automatically update their trust assessments by comparing actual and expected resource usage. Even when human interactions are part of the contract, these feed back into the resource model. In the PDA scenario, even though the trust model is as rich, this automatic analysis and introspection is not appropriate since the value and privacy of confidential information can only be assigned by the user — it is not even possible to deduce information from the user's decisions, e.g. if the user is asked to decide whether to release information to someone else, then neither a 'yes' nor a 'no' means that it was wrong to consult them.

Nevertheless, both the PDA scenario and other contract applications aim to support automatic decision making, and both similarly use rich trust models to allow them to make appropriate cost-benefit analyses, leading to automatic decisions about which interactions to accept. The difference is that purely computational contracts can be completely automatic, while PDA contracts provide partial automation but sometimes need to defer to the user for confirmation.

Contract diversity All PDA interactions answer the same basic question: should Bob be trusted with this information? Although the weights and information values are adjusted by the user, the computational analysis is otherwise identical — beyond that, the user monitors the behaviour of other principals. For generalised contracts, however, far more configurable resource accounting functions are allowed. While this is not a limitation of the contract framework, it is a feature that a PDA application would not make use of, arguing for a specialised implementation even though it fits within the contract framework.

These distinctions show that the PDA scenario has much in common with the general contract framework, but they differ in their core contributions. The trust model for PDA collaboration shows how recommendations can be used to establish electronic analogues of social structures, to control the exchange of personal information. This novel trust model is combined with an explicit cost-benefit analysis to provide an appropriate level of security, without interrupting the user unnecessarily for simple decisions.

On the other hand, the contract model also provides scope for complex trust assessment, but without forcing the choice of a particular model. Instead, the focus there is on expressing low-level interactions in a high-level resource model, against which contract performance is assessed. This assessment depends on explicitly configurable accounting functions, which allow robust, automatic contract assessment tempered with trust.

Thus the PDA scenario fits within the contract framework, but without using its full power. Most importantly, contract compliance monitoring is not fully automatic, but becomes the responsibility of the user, who adjusts the trust assignments appropriately through recommendations. The contracts here govern the right to give out or withhold information or interrupt the PDA owner, and are established whenever information is requested. These are moderated by trust recommendations, and the resources used are intrinsically defined by the interaction itself: they are the data items to be transferred, with their price defined by the PDA owner through the structured information categories. Nevertheless, even this partial automation effectively shields the user from many distractions, providing a significant improvement over the traditional PDA exchange model of confirming every transfer manually. Furthermore, the PDA model presented here provides extra information about the items being transferred, protecting sensitive information from accidental disclosure by others.

Resource measurement and modelling is an essential first step in automatic computation. The effectiveness of this depends on resources being represented accurately and appropriately, to allow a suitable level of control. These need not include only traditional computer resources such as CPU time, but can also incorporate the cost of the user's time, access control credentials, and to a limited extent trust itself. Finally, a PDA collaboration scenario shows the limits of this resource model, in a trust management application in which direct observation of resources gives way to manual configuration, and complete automation is replaced with automated decision support to shield the user from distractions.

Chapter 6

Service Contracts

This chapter shows how the contract framework can support cooperative distributed services, by treating contracts as guarantees of resource availability with explicit penalty clauses. This is validated by using the contract framework for load balancing in a composite event service for a distributed sensor network.

6.1 Events in Contracts

Contract accounting functions monitor their contracts via event notifications as resources are consumed. These notifications can also carry other information about contract performance, such as the special events introduced in Section 4.3 for contract creation and termination.

Extending the range of these events allows contract accounting functions to play a more active rôle in contract management. Not all applications need this; for example, compute servers allow clients to run arbitrary code on a server's computer, and could thus provide contract management in that code. However, for contracts with more constrained actions, the accounting code is the only part of the contract that can be used for this extra control, both because there is nowhere else to incorporate general purpose code and because of the inherent safety of the accounting language with its predictable resource usage profile.

Increased contract complexity, and interaction with outside processes, can make it more difficult to analyse a contract in advance for its expected 'profitability'. However, the threat model for these constrained applications would typically be different from that for general purpose applications; instead of a very cautious approach anticipating attack or abuse by unknown parties, these more specific and more controlled scenarios would imply some extra degree of trust — in the other party, in the nature of the request, or in the contract terms such as conformance to predefined, formulaic accounting templates. This implied trust would clearly not be absolute, but would represent the belief that the contract terms were inherently favourable instead of needing to prove this by stochastic simulation.[1]

Extra events allow the accounting function to stand in for the client as its proxy. This extends the scope of contracts as effective *guarantees* of behaviour. For example, a contract may include explicit penalty clauses for client or server failures. Thus the contract's guarantee would be like a manufacturer's product guarantee — a promise of compensation if there is a flaw, not a theoretical guarantee of perfection. This ability to treat contracts as guarantees makes it simpler to design resource control systems in situations in which a purely traditional resource model would not receive enough information to monitor contract performance.

Contracts need not only control competitive environments; they can also serve as the control system for a collaborative network, which provides a service to outside users. In this case, contract profitability would not ultimately be associated with financial resources, but rather with resources in a constructed economy used only to prioritise cooperative actions.

A collaborative approach retains the notion of cost accounting, but makes more assumptions about others' abilities. The contracting framework does still provide a degree of resilience, but only to failures measured by the accounting functions. Thus, for example, an application might enable the detection of an overloaded node with high data lag, but not of a rogue node providing incorrect data.

A contract-based approach allows both competitive and cooperative environments to monitor and control activities, using cost as a metric and with the contracts as effective behaviour guarantees. To demonstrate this, the following section describes a distributed service for composite event detection, while Section 6.3 shows performance results when contracts are used to assist in the detector placement within an unreliable network.

[1] By the same token, a more complex accounting function would make it harder for a client to decide whether it was being applied accurately and correctly, so the client would also need enough intrinsic trust in the server to accept the contract.

6.2 Event Composition in Distributed Systems

Event-based systems allow large-scale, reliable distributed applications to be built, structured around notification messages sent between their components when something of interest occurs. However, especially in large-scale applications, the recipients might be overwhelmed by the vast number of primitive, low-level events from many sources, and would benefit from a higher-level view. *Composite events* can provide this view, by automatically detecting when a pattern of events occurs and then generating only a single notification message.

For example, a company might use an event system to coordinate its internal network services and databases across a number of autonomous departments. Thus the sales department would notify the warehouses of the availability of its ordering database and also when new orders were placed, and the warehouses would in turn notify suppliers when more inventory was needed. A composite event service would then allow more sophisticated requests, such as management requests for notification of all orders over £10 000 from new clients, or of database failures lasting longer than 15 minutes.

In a publish/subscribe event system, events are the basic communication mechanism. Components in the system are identified as event sources, event sinks, or both: event sources *publish* messages, which are in turn received by those event sinks which *subscribe* to them. The event system is then responsible for efficiently routing the messages to their recipients. A publish/subscribe system may also provide extra features, such as service guarantees (e.g. guaranteed delivery, event logging, message security) and expressive subscriptions; in topic-based services, each event is published with an associated topic, and is received only by sources which subscribe to that topic[2], while content-based services allow subscriptions based on the contents of an event's attributes, e.g. only order notification events of more than £10 000.

Publish/subscribe systems also often have a messaging infrastructure consisting of a network of intermediate *broker nodes*, which are used to route messages efficiently between publishers and subscribers. These broker nodes can make more efficient use of bandwidth than direct point-to-point notification between publishers and subscribers, because only one copy of each message needs to be sent from each broker to the next, and the final brokers can then use high-bandwidth local area networks for the final notifications.

Lastly, the indirection of a publish/subscribe network provides extra flexibility

[2] or a parent topic, in a topic hierarchy

and anonymity for publishers and subscribers. Because subscriptions are based on topic and content, not the publisher's identity, the publisher is effectively anonymous (in the sense that their identity need not be known) — there need not even be only one publisher for each topic. Similarly, the publisher does not need to know the identities of its subscribers. This allows publishers and subscribers to be added and removed independently of each other, improving system stability and reducing the need to reconfigure as the flow of information changes.

Composite event support need not necessarily be built into an event system directly. Instead, it can be an independent extension to an existing event service, by providing a proxy interface through which the application can access the event services. For publishing and subscribing to primitive, non-composite events, the proxy interface simply redirects requests to the original event interface. The other requests are instead processed by the composite event service, which disguises composite events by encapsulating them into primitive events with special type names or similar identifiers. Composite event publications and subscriptions are therefore translated into appropriate primitive event requests for the disguised events. The underlying event system is also used to coordinate the operation of special *composite event detection broker* nodes in the network, which host composite event detectors for external subscribers.

6.2.1 A Language for Composite Events

The composite event detection framework presented in this section includes a language for expressing event patterns. This language was designed to satisfy three goals: compatibility with existing regular expression syntax, potential for distribution of common sub-expressions, and ability to reflect the underlying publish/subscribe system.

Regular expressions provide a well-known syntax for defining words and patterns in strings of text, and are an essential tool in the design of compilers [2]. The composite event language extends the standard regular expression operators, with operators for timing control, parallelisation and weak/strong event sequencing. (Section 6.2.2 explains interval timestamps for composite events, which induce two event orderings.) This language is minimal in that there are no redundant operators, and furthermore, it includes basic regular expressions as a subspace.

Distribution allows parts of a complex expression to be distributed to different computers in the event network. This allows commonly occurring subexpressions to be reused, improving efficiency. Subexpressions can also be positioned close in the network to the sources of the events they detect — this saves bandwidth, and can indirectly reduce latency too by reducing network congestion. The composite event language structure therefore needs to have a structure which supports factorisation and expression reuse.

For effective distribution, expressions must also have bounded computational needs — otherwise it would be unsafe for brokers in the network to host distributed expressions because of the risk that these would have unsustainable resource needs.

Reflecting the underlying publish/subscribe system allows the composite event framework to take full advantage of extra features when they are available, such as content-based event filtering. However, by isolating these features from the core composite event framework, it can still remain useful even when they are not available.

The language consists of the following structures:[3]

Atoms. $[A, B, C, \cdots \subseteq \Sigma_0]$. Atoms detect individual events in the input stream. Here, only events in $A \cup B \cup C \cup \ldots$ will be successfully matched. Other events in Σ_0 will cause a failed detection, and events outside Σ_0 will be ignored. We abbreviate negation using $[\neg E \subseteq \Sigma]$ for $[\Sigma \backslash E \subseteq \Sigma]$, and also write $[E]$ instead of $[E \subseteq E]$. (Negation ensures any other events in Σ will stop the detection, such as timeouts or stopper events.)

Concatenation. $\mathcal{C}_1\mathcal{C}_2$. Detects expression $\mathcal{C}_1$ *weakly* followed by $\mathcal{C}_2$.

Sequence. $\mathcal{C}_1;\mathcal{C}_2$. This detects expression $\mathcal{C}_1$ *strongly* followed by $\mathcal{C}_2$. Thus $\mathcal{C}_1$ and $\mathcal{C}_2$ must not overlap in a sequence, but they may in a concatenation.

Iteration. $\mathcal{C}_1^*$. Detects any number of occurrences of expression $\mathcal{C}_1$. If $\mathcal{C}_1$ detects a symbol which causes it to fail, then $\mathcal{C}_1^*$ will fail too. (So $[A][A \subseteq \{A,B\}]^*[C]$ would match input AAC but not $AABC$.)

Alternation. $\mathcal{C}_1 \,|\, \mathcal{C}_2$. This expression will match if either $\mathcal{C}_1$ or $\mathcal{C}_2$ is matched.

Timing. $(\mathcal{C}_1, \mathcal{C}_2)_{T_1=\text{timespec}}$. The timing operator detects event combinations within, or not within, a given interval. The second expression $\mathcal{C}_2$ can then use T_1 in its event specification.

[3] This formal language specification and the automata derivation were published in a paper for the Middleware 2003 conference, in collaboration with Peter Pietzuch and Jean Bacon [82].

Parallelisation. $\mathcal{C}_1 \,\|\, \mathcal{C}_2$. Parallelisation detects two composite events in parallel, and succeeds only if both are detected. Unlike alternation, any order is allowed, and the events may overlap in time.

In this model, atoms act as the interface between the composite event service and the underlying event system. Not only do they allow primitive events to be included in composite event expressions, but they also provide a mechanism for composite event expressions to be distributed over a network and benefit from the extra features of the underlying event system.

Whenever a composite event is generated, it is encapsulated into a new event, and sent to its recipients using the publish/subscribe system. This event, c_1, could also be received by another composite event detector as part of an event atom, provided that the space of acceptable inputs allowed it ($c_1 \in \Sigma$). Furthermore, the new detector could use the publish/subscribe system to impose constraints on its input if the subscription model supported them, e.g. ensuring that c_1's constituent primitive events all referred to the same person (if the publish/subscribe system supported parameterized attribute filtering).

The following examples show how the core composite event language can be used to describe composite events. Let A be the events corresponding to 'a large order is placed', let B be 'the ordering database is offline' and C 'the ordering database is online'. These would each represent an expression in the original event subscription language; for example A might stand for `OrderEvent(value>10000)`.

1. Two large orders are placed within an hour of each other: $([A], [A \subseteq \{A, T_1\}])_{T_1 = 1\,\text{hour}}$
2. A large order is placed but the ordering database is offline: $[B][A \subseteq \{A, C\}]$

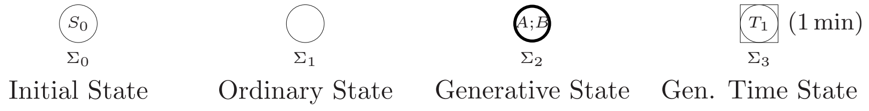
6.2.2 Composite Event Detection Automata

Expressions in the composite event language are automatically compiled into automata, similar to the finite state machines used to detect regular expressions. There are differences though, because composite event detection automata have a richer time model and are inherently nondeterministic.

Interval timestamps Events can be ordered by the time at which they occurred. However, in a large-scale distributed system, event timestamps may have a degree of uncertainty associated with them. Furthermore, composite events naturally occur not at an instant but over an interval of time; incorporating the uncertainty of all of the constituent events, this runs from the earliest possible start time of the first event until the last possible end of the last event). Events are therefore sequenced using an *interval timestamp.* A partial order $<$ shows which events definitely occurred before others and is used for the strong event sequencing operator — but with this operator, some event times may not be comparable when their intervals overlap. (Either it is unknown which occurred first, or they might be composite events which really overlap in time.) There is also a total order $\prec$, used for the weak concatenation operator, which extends the partial order using a tie-breaker convention to allow all events to be placed in some consistent order.

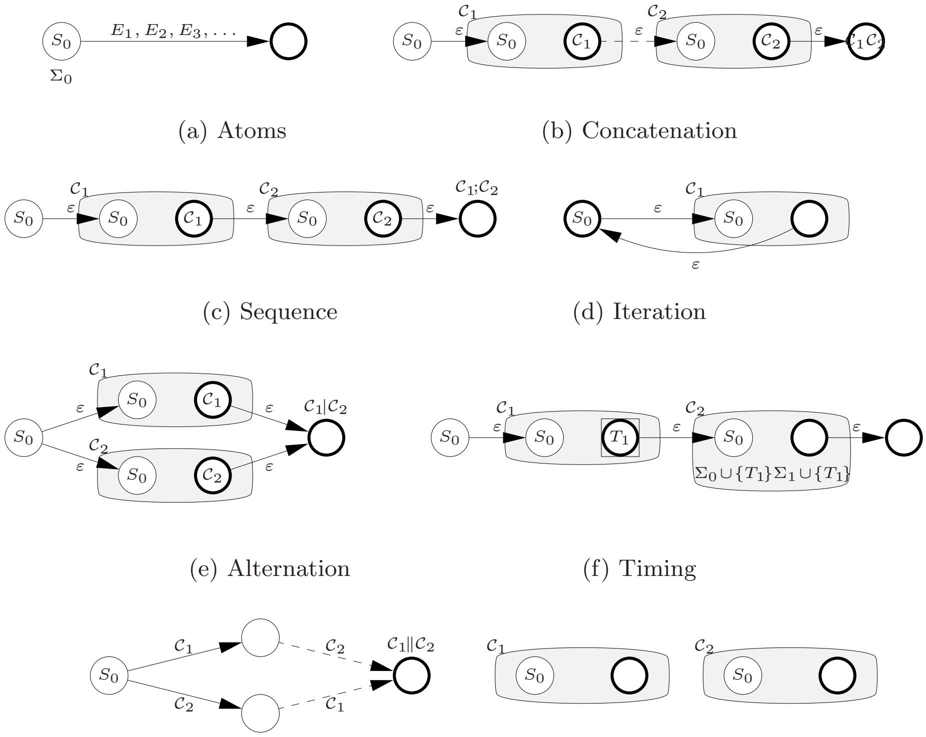
Nondeterminism Conventional finite state automata can always be converted from non-deterministic to deterministic form. However, the composite event automata are inherently nondeterministic, because each state needs an associated timestamp to support strong and weak event sequencing; converting the automata to a deterministic form would require multiple timestamps per state.

Each state has an input domain, the family of events it can match. When in a given state, the automaton processes only those new events that lie within the state's domain (as opposed to finite state machines which conventionally receive all symbols in a text string). The diagram below shows the four types of state: an initial (ordinary) state, an ordinary state, a generative state for a composite event of type '$A; B$', and a generative state for a time event, which will be generated automatically after the time interval and can feature in the input domain of later states. The input domains are $\Sigma_0 \ldots \Sigma_3$.



States are connected by strong or weak transitions: strong transitions are represented by solid lines and require that the next event detected must strongly follow the previous event in the interval time model, while weak transitions allow overlapping events and are shown as dashed lines. Each transition is also labelled with the events that will cause it to be taken.

Expressions in the event composition language are translated into automata recursively, beginning with the simplest, innermost expressions and working outwards according to the constructions shown in Figure 6.1.


S_0
$E_1, E_2, E_3, \ldots$
Σ_0
$\mathcal{C}_1$
$\mathcal{C}_2$
ε
$\mathcal{C}_1\mathcal{C}_2$
$\mathcal{C}_1;\mathcal{C}_2$
$\mathcal{C}_1|\mathcal{C}_2$
T_1
$\Sigma_0 \cup \{T_1\}\Sigma_1 \cup \{T_1\}$
$\mathcal{C}_1\|\mathcal{C}_2$

(a) Atoms

(b) Concatenation

(c) Sequence

(d) Iteration

(e) Alternation

(f) Timing

(g) Parallelisation

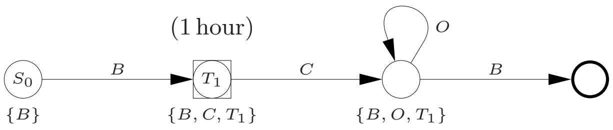
Figure 6.1. Construction of composite event detectors

However, if a subexpression is to be distributed to another broker node, it is generated independently, and its placeholder in the overall detector is replaced with a single transition, representing a subscription to the composite event generated by the subexpression.

For example, if the ordering system goes down twice within an hour, the sales department might want to double-check any orders made during the uptime before the second failure. As before, B means 'the ordering database is offline', C is 'the ordering database is online' and let O represent all new order notifications. The expression to be detected is then

$$\mathcal{C}_1 = ([B],\ [C \subseteq \{B, C, T_1\}]\, [O \subseteq \{B, O, T_1\}]^*\, [B \subseteq \{B, O, T_1\}])_{T_1 = 1\,\text{hour}}$$

This is compiled into the following automaton, according to the rules above:



This novel composite event detection framework demonstrates how this can be implemented as a generic middleware extension, independently of the underlying event system. The composite event service does not require any special features of the underlying event middleware, but can take full advantage of its expressive power in detecting patterns of events. A decomposable core language allows these expressions to be factorised into independent subexpressions and distributed across the network for efficiency and redundancy, where they are compiled into detection automata. Finally, this distribution is inherently safe because the constrained structure of composite event expressions limits the detectors' resource consumption.

6.3 Contracts in Event Composition

Detector distribution is important for the performance of composite event detection, as outlined in the previous section. This is demonstrated here with performance results from a sensor network application, which uses the contract framework for load balancing.

The sensor network consists of a number of environmental sensors in a building, all connected to a publish/subscribe network with 100 event brokers. Sensors generate events periodically or when they detect a change in the environment. These event notifications are then routed to their subscribers via the event brokers, following the model of Java Message Service (JMS) implementations [97]. Composite event support is provided by enabling each event broker to host composite event detectors, in addition to its basic event forwarding behaviour.

Communications within the sensor network are assumed to be performed on a best-effort basis, over an unreliable network [100]. In particular, the network links have only a limited bandwidth available and limited storage buffers for holding unprocessed data; data packets (events) which overflow these buffers are discarded. This unreliability need not be a major problem for the network, provided that its effect is randomly distributed; for example, if events are repeated periodically, then all subscribers will eventually receive status updates for long-term changes even though transient changes might be lost.[4] This is appropriate for sensor networks such as those which monitor the status and usage of a building: although changes in state such as room temperature need to be monitored, occasional data loss is acceptable, simply resulting in a less

[4] Events from a sensor can also include a timestamp or sequence counter. This would allow lost events to be inferred for monitoring purposes, even if their contents were lost.

responsive system and not in any real damage. Nevertheless, the best effort system would still provide better responsiveness on average than a guaranteed delivery system designed to generate all events slowly enough not to exceed the network's peak rate under any circumstances.

Guaranteed delivery does still have a rôle in the sensor network, for emergency use. By giving emergency traffic priority over all other data, the network can be engineered to ensure that it has enough capacity to guarantee the delivery of all emergency events, even faced with multiple emergencies, e.g. fires in different parts of the building, coupled with a burst water main. Any unused capacity can then be used for best-effort delivery of other sensor events.

Composite event detector placement within the sensor network affects the reliability with which event patterns are detected and the results made known. Although detection will always be unreliable, good detector placement can reduce data loss and thus improve the system's responsiveness. For example, a detector on an overloaded broker or on one with poor links to its event sources would lose more data than one closer to the sources in network terms. As there may be many, changing, event sources for a given subscription, this distance metric is a theoretical measure only. However, it can be assessed in relative terms by comparing the outputs of two similar detectors to each other.

To test the effect of detector placement policies, a sensor network was created as described above, and populated with 50 sensors as publishers and with 25 composite event subscribers. Each subscription consisted of a random concatenation of five primitive events, e.g. $[A][B][D][D][C]$. For the composite event subscriptions, *detectors* were constructed automatically, initially at random broker locations in the network; the configuration for a single composite event type is shown in Figure 6.2. The subscription expressions were also automatically factorised where possible, to improve subexpression reuse. Finally, for analysis purposes and to split the load, each composite event detector was replicated onto two brokers, and subscribers were assigned a detector at random. Together with the replication, a *monitor* was added for each expression, collocated with an existing subscriber and subscribing to both detectors to support performance comparisons, as illustrated in Figure 6.2. In this diagram, ordinary subscriptions are shown as dashed lines, while the solid lines are monitoring contracts combined with subscriptions. The detectors for this subscription are located on the shaded broker nodes; these detectors in turn subscribe to other primitive and composite event sources, which are replicated themselves and have their own monitors.

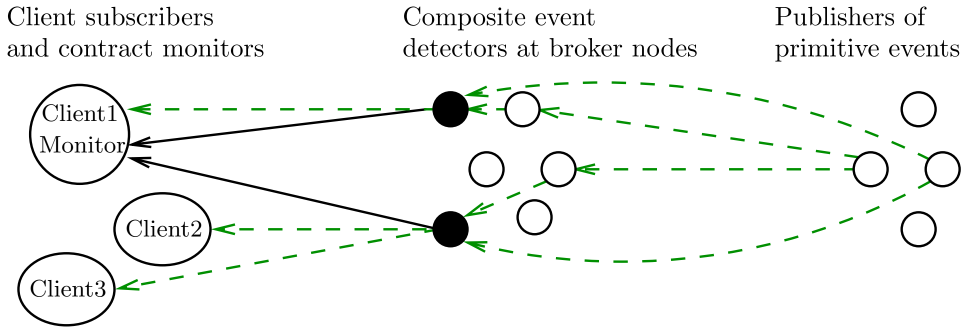


Figure 6.2. Contracts and subscriptions for a single composite event subscription type

This testbed was exercised with two different detector placement policies: naïve placement and contract-based load balancing.

Naïve placement positioned the detectors randomly throughout the broker network. Each monitor detected if either of its detectors had failed entirely, but would not migrate detectors to new locations otherwise.

Contract-based detection treated each broker as a server for hosting event composition detection contracts, with the monitors as clients requesting these services. The servers here tried to satisfy all requests by creating detectors for all composite event expressions contracted for by the monitors, while the monitor clients assessed contract performance in terms of the total number of detections and the number of failed detections, in order to calculate contract profitability. Failed detections were identified whenever a monitor received a particular composite event notification from only one of its detectors but not the other, within a given time window. If a broker performed poorly (relative to its opposite number) then it was replaced with another of higher apparent trustworthiness, if possible. Brokers could either have pre-initialised trust values, allowing an operator to guide detector placement on a per-type basis, or assess trust values purely dynamically. Profitability analysis effectively defines a target error rate that is acceptable, below which the monitor will not actively seek out new brokers for its expression.[5] This is implemented by the following contract accounting function:

```
class Accountant(resources.ResourceContract):
   def processResourceAtom(self, atom, imports):
      if atom.type == resources.eventGenerated:
         rate = 1
```

[5] This strategy could be altered to speculatively test a new broker occasionally, irrespective of the current brokers' performance.

```
    elif atom.type == resources.eventFailure:
       rate = -20
    else: return [] # Charge only for the above types
    return [ResourceAtom(resources.money, '£', '',
                         rate*atom.quantity,
                         atom.startTime, atom.endTime)]
```

The contract monitors essentially act at the type owners for their corresponding composite event types, since they are responsible for controlling the detector-hosting brokers which publish all events of those types. These monitors are currently identified by sending messages on a special administration topic — this is handled transparently by the composite event interface library. If no reply is received, a new contract monitor is established for that type. This may occasionally result in multiple masters for a particular composite event type; this will not lead to failures, but could cause more events to be transmitted than necessary, so the monitors also subscribe to other monitors' replies on the administration topic. If one monitor discovers another on the same topic, they negotiate a handover of subscriptions and the unneeded monitor is then shut down.

The contract monitors and the composite event detectors together form a single cooperative trust network, which operates in a peer-to-peer fashion. This network acts to regulate itself by migrating underperforming detectors to other nodes, thus performing load balancing when brokers are overloaded.

The performance results of the composite event detection experiment are shown in Figures 6.3 and 6.4, which compare naïve and contract-based detector placement while 20 000 primitive events were published, in terms of the number of composite event detection failures and the relative frequency of these failures — lower values are better in both graphs.

These results show that contract-based placement reduced the overall number of failures by 20% (from 2009 to 1602), and the relative rate of failure by 27% (from about 3.9% to about 2.8%). This second measure of improvement is higher than the first because it also takes into account the concomitant increase in the number of successful messages together with the decrease in failures.

The early portion of each performance graph also shows a period where no failures are detected, and a brief interval where the naïve solution seems to perform better. The initial absence of failure detections has two causes: there is a period where few composite events are detected as events propagate and the detectors begin to form partial matches of event patterns, and an intentional

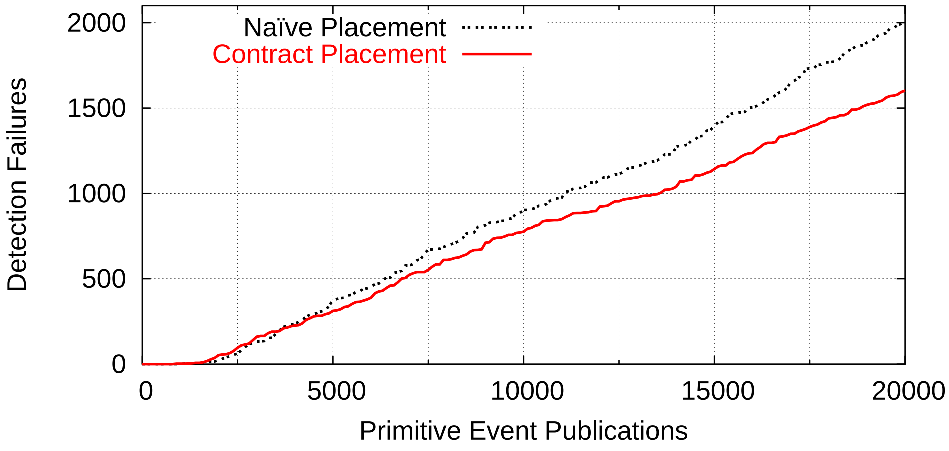


Figure 6.3. Graph of detection failures over time, for naïve and contract-based detector placement

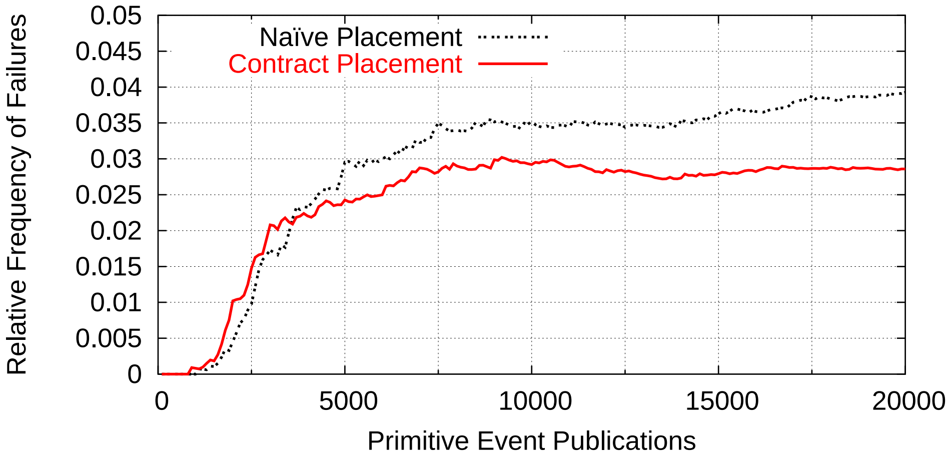


Figure 6.4. Graph of ratio between successful and failed composite event receipts

lag is also inserted into the failure calculation code which waits for a predefined period of time before labelling an event detection disparity as a failure. The brief interval of performance inversion is caused by the first wave of detector migrations; in the current implementation, any partial composite event match notifications that were still in transit when the detector migration occurred are lost (to the composite event detection framework), so detector migration can introduce additional transient detection failures. Nevertheless, this effect could be mitigated with a more sophisticated detector migration protocol, at the cost of sending extra event notifications. Overall however, the contract-based solution shows a clear and consistent increase in performance over the naïve approach.

Thus a contract model can assist in the development and implementation of distributed services and peer-to-peer networks. In these applications — whether competitive or collaborative — contracts' accounting functions act as explicit guarantees of behaviour, incorporating penalty clauses to be used in the event of failures. The effectiveness of this approach is demonstrated using a distributed composite event service, in which data loss on a best-effort network was reduced by 20% by using cooperative contracts to guide detector placement.

Chapter 7

Implementation and Results

This chapter validates the computational contract framework in the implementation of a compute server. Section 7.1 presents a standalone server which shows that the accounting model successfully prioritises more lucrative contracts, and that resource-based trust modelling increases profitability over simple trust. Section 7.2 extends the server with support for trust recommendations, showing that recommendations can bootstrap a corporate network, but that subjective trust assessments are then adjusted over time to reflect the quality of service actually available. Finally, cheating principals test the resilience of the trust system to attack.

7.1 Compute Server Implementation

A compute server lies at the heart of a commercial Grid service. To be effective and useful, it needs both to ensure that its tasks are profitable, and to perform them faithfully so that its users trust it. This section shows how the compute server described in Section 4.3 is implemented.

7.1.1 Architecture and Implementation Design

Figure 7.1 illustrates the major components of the compute server, which has two separate threads of execution — for contract computation and for communications. In the computation thread, the *contract scheduler* decides which contract should next receive more resources, and then allows it to execute for a time. After the contract has waived or exhausted its resource allocation (e.g. at the end of its time slice), control is passed to the *accounting modul*e, which

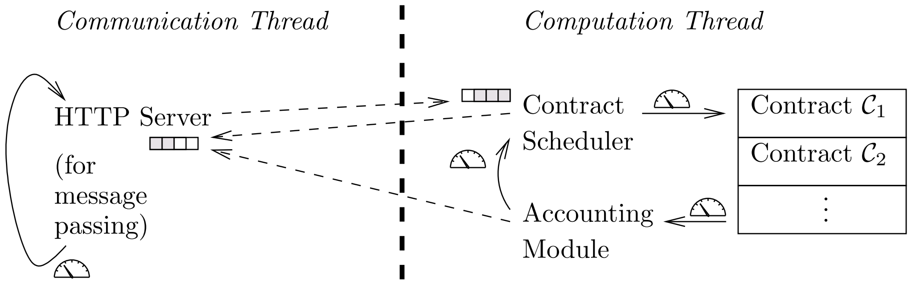


Figure 7.1. Threads of execution in a compute server

calculates the payment expected from the contract's updated resource tallies, before returning control to the contract scheduler again. At the same time, an *HTTP server* operates in the communication thread, listening for contract messages from others and taking responsibility for sending out contract messages such as payment requests. Within each thread, the flow of control is shown using solid arrows.

Communication between the two threads takes place indirectly via two message queues (shown as ▭▭▭ in Figure 7.1). The contract scheduler and accounting module place messages that need to be sent into a queue for the HTTP server, while messages received by the HTTP server are placed in the contract scheduler's queue, signified by dashed arrows to the corresponding queues. These producer/consumer queues ensure that neither thread blocks the other for an appreciable length of time.

Accounting of resource usage takes place at a number points in the compute server application, indicated by meter icons (⏲) in Figure 7.1. At each of these points, the extra resources used since the previous measurement are computed and added to the tally of the appropriate contract. Resources used in executing a contract and in its accounting both contribute to the contract's resource tally, while those used by the contract scheduler contribute to its tally. Periodically, the contract scheduler's resource usage is split between the current contracts, to be paid for by their accounting functions, as it represents collective system overheads that cannot effectively be attributed to a particular task.

The communication thread also accounts for the resources that it uses on behalf of each contract, in sending or receiving messages. Both the messages received and the resources used are communicated to the contract scheduler through its queue, ensuring that only the computation thread is responsible for adjusting resource consumption.[1] Some communication resources may not be attributable

[1] The communication thread can nevertheless read resource consumption values for con-

to a particular contract, such as those used if a contract is abandoned after initial negotiations. These are then attributed to a special dummy 'null contract', which can either be subsidized by the server as part of the overhead of doing business (to be offset against net profitability as described in Section 3.3.1) or else periodically divided proportionally between the active contracts, in the same way as the contract scheduler's overheads.

The compute server is implemented using the interpreted Python language for all its operations, including contract management and contract execution. Although reimplementing the code in a compiled language such as C++ would undoubtedly reduce the computational overheads of the contract framework, the overall effect on relative performance would be small since the Python-based contracts themselves have proportionately similar overheads. Furthermore, although contract accounting functions could be interpreted more efficiently with the help of a dedicated parser, they are written using a valid Python subset and can therefore be evaluated directly using the standard Python interpreter.[2]

The tasks of each component are as follows:

Contract Scheduler The scheduler ensures that tasks are given the resources promised in their contracts, if possible and if they have paid the amounts expected. It is also responsible for processing contract messages and updating the contract state accordingly. (The HTTP Server performs any complex calculations and signature verification; the contract server simply integrates this information.) If a contract is due to provide payments, the amounts are expressed as resources and stored in a `resourcesToBeProvided` field, together with timestamps representing when each payment falls due.

To ensure consistency, contracts whose participants have paid the full resource amount expected are always allowed to be scheduled when resources are due to them under the contract terms. For other contracts, the participant profitability tests of Chapter 4 are used to exclude unrewarding contracts and prioritise those which are profitable, when providing both the minimum resource allocation guaranteed and any extra resources that are available.

tracts and principals, and thus avoid wasting resources on principals whose contracts have been found to be unprofitable.

[2] Strictly speaking, the current implementation should check that accounting functions do indeed conform to the Python subset specified in Table 4.3 and automatically ignore them if they do not; accounting functions that parse correctly are necessarily be safe to execute. However, this extra protection is not needed in the tests below, which test the contract framework's robustness through attacks such as non-payment and deceptive contract specifications.

Accounting Module This module assesses the compliance of each contract with its terms, and determines any payment required. To do so, it associates extra information with each contract, representing the current state of the contract's accounting function and resources used under the contract but not yet accounted for (as `resourcesUnaccounted`). The accounting module also updates the tallies of resources received and still to be received by the contract from the compute server system, such as CPU resources and bandwidth, and resources provided and still to be provided to the system by the contract, such as payments.

HTTP Server The web server which receives contract messages is also responsible for checking message signatures on receipt, and for signing or countersigning outgoing messages to other compute servers. For simplicity, the current implementation assumes that servers have another mechanism for establishing pairwise shared symmetric keys when needed (with the help of public and private key pairs), and servers then use these keys to encrypt the contract data *en route*. Resources used in this encryption and decryption are automatically reported back to the contract scheduler for later accounting. Payment messages are currently only processed internally with a specified resource overhead, instead of actually passing them to an online micro-payment service in order to credit a real account [99].

Finally, the HTTP server is also responsible for deciding which new contracts to accept, based on past participant profitability and profitability predicted from the contract terms.

Both the accounting module and the contract scheduler interface with the underlying operating system exclusively through a `ResourceSystem` class instance with three methods: `giveResources`, `enterFrame` and `exitFrame`. `giveResources` is used by the contract scheduler to request that a contract be given a specific allocation of resources. In the current implementation, this entails a cooperative call to the Python code implementing the contract, but this mechanism could equally be used to control an existing operating system scheduler or a dedicated preemptive CPU scheduler. (This is discussed in more detail in Section 7.3.)

The `enterFrame` and `exitFrame` methods act as checkpoints for calculating actual resource consumption and passing this information on to the accounting module to integrate it into the appropriate contract. These methods maintain a contract stack, with the innermost contract responsible for the resources used at any given moment; this nesting ensures that there is always some contract responsible for all resource usage, so that no resource usage goes unaccounted

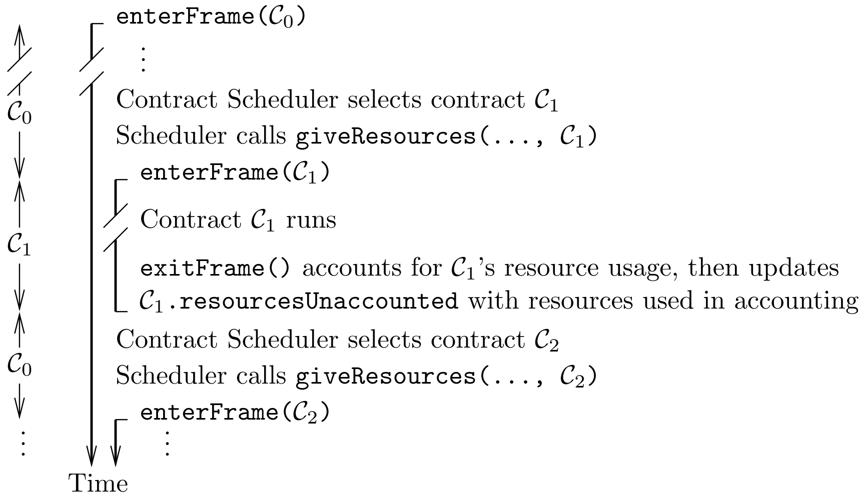

enterFrame($\mathcal{C}_0$)
⋮
Contract Scheduler selects contract $\mathcal{C}_1$
Scheduler calls giveResources(..., $\mathcal{C}_1$)
enterFrame($\mathcal{C}_1$)
Contract $\mathcal{C}_1$ runs
exitFrame() accounts for $\mathcal{C}_1$'s resource usage, then updates
$\mathcal{C}_1$.resourcesUnaccounted with resources used in accounting
Contract Scheduler selects contract $\mathcal{C}_2$
Scheduler calls giveResources(..., $\mathcal{C}_2$)
enterFrame($\mathcal{C}_2$)
⋮
$\mathcal{C}_0$
$\mathcal{C}_1$
$\mathcal{C}_0$
⋮
Time

Figure 7.2. A time-line showing how resources are attributed to contracts.

for. This is illustrated in Figure 7.2 which shows a time-line of contract execution, and the contract to which resource usage is attributed at each moment. Periodically, the scheduler exits and re-enters its own resource frame, so that its resource usage can be accounted for too.

7.1.2 Tests and Results

Automatic tests are required to test the effectiveness of the contract framework, assessing the general overheads of contract support, the effectiveness of the accounting model in identifying and prioritising appropriate contracts, and the ability of the trust model to ensure overall profitability of the system in the face of clients defaulting on their payments.

The tests of the contracting framework are performed using a network simulation similar to that described in Section 6.3. However, while that network provided unreliable, best effort delivery (such as that provided by UDP in TCP/IP networks), this network simulates reliable delivery over variable bandwidth links (as with TCP).

In this environment, a single machine simulates a large number of networked nodes; each node's simulator runs as an independent process, operating a single contract scheduler and its corresponding communication thread. All inter-contract communication takes place through these threads, both for contract management messages and for messages generated in the execution of the contract. This allows all communication resources to be monitored directly by the contracting system, while CPU resource usage is assessed by each contract scheduler using standard operating system calls. Cooperative multitasking allows the contracts themselves to assess their own resource usage through the same mechanisms, and the corresponding entry and exit points (or yield points) are used as their mechanism for sending and receiving messages.

Contracts can send messages to other contracts, and also special messages which have meaning for their contract scheduler. For example, returning the event `events.Sleep(10)` requests that the scheduler not provide the process with more CPU resources for the next 10 seconds (unless new messages are received through the communication thread), while an `events.TerminateMe()` message requests that the contract be cleanly terminated, and `events.PercentageDone(0.80)` signals that the contract's code is 80% complete.

The network simulator associates a timestamp with each message sent and received, appropriately adjusted for delays introduced by the network. Each node simulator maintains an internal timestamp reflecting the time that has elapsed for its processes. When a node receives a message, that message is then held in a queue, until the simulated time of the node reaches the delivery time of the message, when it is delivered to the appropriate contract. To ensure consistency, the simulation framework also ensures that no node advances its clock substantially beyond the others'.

Testing of the contract framework aims to measure the overhead it introduces, and the effects of introspection and continual 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, a control node establishes a PBIL optimisation server at one node, which uses contracts to establish clients on remote nodes.

Population-Based Incremental Learning (PBIL) [81] is an optimisation technique which is well suited to distributed operation. It uses a random process to generate conjectured optimal parameters for the function being analysed, expressed as bit sequences. The parameters that generate the best outputs are then used as seeds to bias the random process in the next iteration to generate similar parameters in future. PBIL can take advantage of distribution in two ways: firstly, a function may have a number of local optima which make it difficult to locate the global optimum, therefore a number of independent searches should be performed. Secondly, the calculations within each search can be spread over a number of computational nodes; each node generates and tests its own set of random parameter values, but also periodically notifies the other nodes of its best results, which they then incorporate into their populations.

In this test, 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 instead of establishing them by contract, 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, in which the expected profitability of contracts is automatically assessed from their accounting functions, and then compared to actual profitability. 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 is not 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 a conservative resource allocation strategy, 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% for trust modelling and 132% for blind trust, showing that effective trust modelling can improve performance. These values are significantly higher than in the earlier test because these contracts were designed to be potentially extremely profitable.

7.2 Trust Integration

The second set of tests demonstrates that trust recommendations are effective for establishing and maintaining appropriate trust in a corporate environment. In these tests, two branch offices are simulated with very good local interconnections but with an unreliable link between the offices. On each side of the link are five computers, each running a compute server, named a_1 to a_5 and b_1 to b_5.

The first of these tests uses recommendations to bootstrap trust values in the network. A manager issues recommendations for each computer in the local office, and a user u_1 (who has strong trust in the manager) applies these recommendations to compute its trust assessment. These recommendations are issued in pairs; the manager m_1 not only recommends a_5 to u_1 (in the recommendation given below), but also recommends u_1 to a_5.

```
Recommendation( Recommender = "m1",
                Recommended = "a5",
                Degree = (5.0, 5.0, 3.0, 3.0),
                Limits = (5.0, 0.20))
```

When the user performs computations on the servers, it selects servers which are profitable and have a high expected return on investment $\frac{t_{ar}}{t_{ao}}$. The likelihood of selecting a server is biased according to the expected benefits, so that a server whose computations resulted in 20% average profit would be selected twice as frequently as one generating 10% average profit. The rationale for this user behaviour would be to spread the contract risk between servers, and at the same time gather extra information about service reliability levels, while still

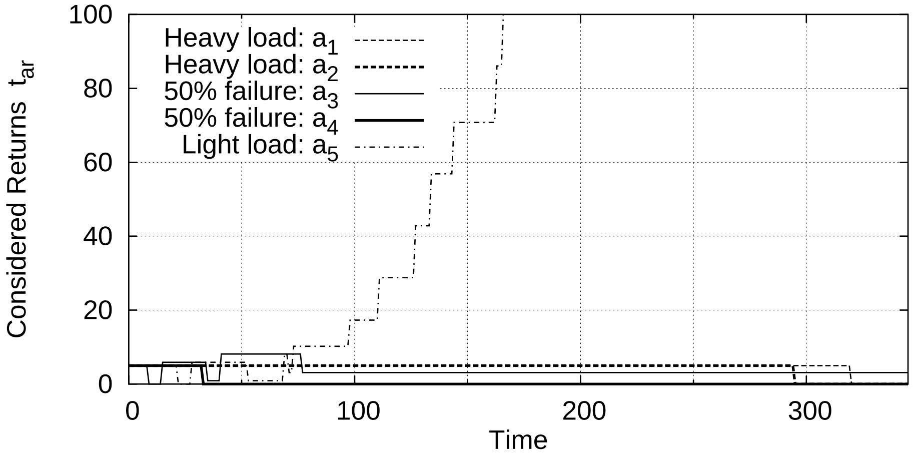


Figure 7.3. Graph of servers' actual returns over time t_{ar}, for user u_1

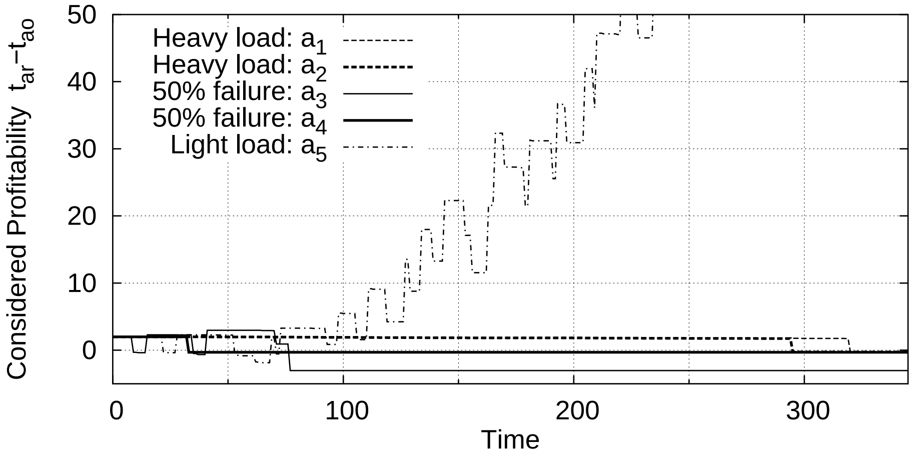


Figure 7.4. Graph of server profitability over time $t_{ar} - t_{ao}$, for user u_1

ensuring only profitable contracts were undertaken. The user is also allocated a constant-rate resource budget with which to pay for contracts; this leads to the gross contract returns and profitabilities shown in Figure 7.3 and Figure 7.4, calculated after considering the effect of recommendations. For these graphs, two of the servers (a_1 and a_2) are assumed to be heavily loaded and refusing contracts, two (a_3 and a_4) are heavily loaded but failing to complete 50% of contracts, and one (a_5) is lightly loaded and performing all its contracts correctly.

At first, the recommendations account for almost all of u_1's trust in each server; as productive contracts with a_5 proceed, u_1 begins to trust a_5 in its own right and the weight of the recommendation is reduced. The scheme described in Section 4.3.2 is used to ensure that a_5 gains at least 5% of the resulting trust even when its apparent profitability arises only from recommendations. How-

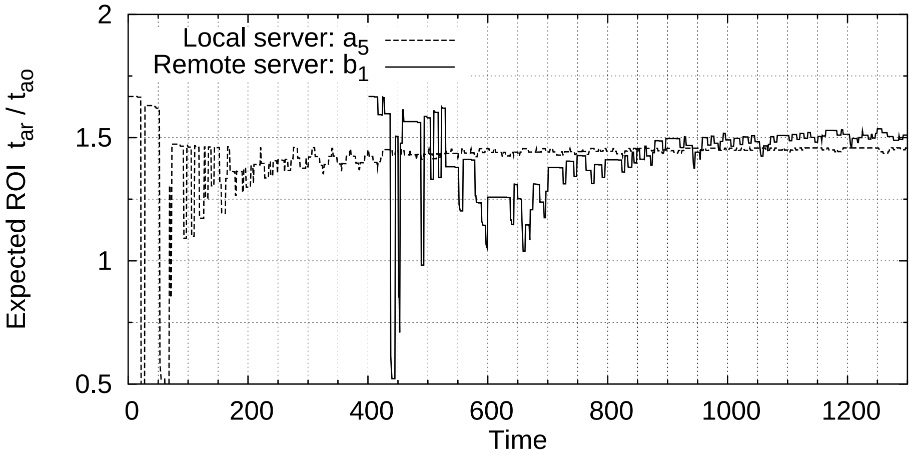


Figure 7.5. Graph of expected returns for server b_1 over time

ever, when contracts with a_1 and a_2 are attempted, the requests are refused. The overheads of the contract requests consume small quantities of resources, ultimately proving the servers to be unprofitable, as shown by the slowly decreasing dashed lines in Figure 7.4. Ultimately these drop to zero when the limits of the manager's trust recommendation are reached. (This also explains the drops in considered contract returns in Figure 7.3 at time 295 and time 320, since these apparent returns stemmed from the recommendations.) Finally, the unreliable server a_3 performs two contracts successfully before proving unprofitable, while a_4 is immediately abandoned after the first contract. Interactions with a_3 eventually lead to a slight net resource loss, but this is strictly limited because the user u_1 is programmed to hazard the value of a contract in advance once it has been agreed upon, and not expect any return until after it is completed.

Suppose the manager then at time 400 issues a strong recommendation to u_1 for a lightly loaded computer b_1 on the other side of the link which charges less for its resources than a_5; this is integrated into the user's trust assessment so that the user's trust in b_1 goes from $(t_{er}, t_{ar}, t_{eo}, t_{ao}) = (0, 0, 0, 0)$ to $(10, 10, 5.5, 6)$, based on u_1's experience of the manager's past recommendations. Figure 7.5 illustrates how the user's expected return on contracts with b_1 changes over time; initially, the unreliable link between the branch offices causes 65% of the computational results from contracts to be excessively delayed in transit giving $\frac{t_{ar}}{t_{ao}} \approx 1.25$, but after time 650, the link quality increases to 90% and the assessment of b_1's expected return on investment also rises to approximately 1.5, performing better than the consistent but more expensive a_5 at 1.45.

Finally, a second user u_2 is introduced at time 1400, recommended by the same

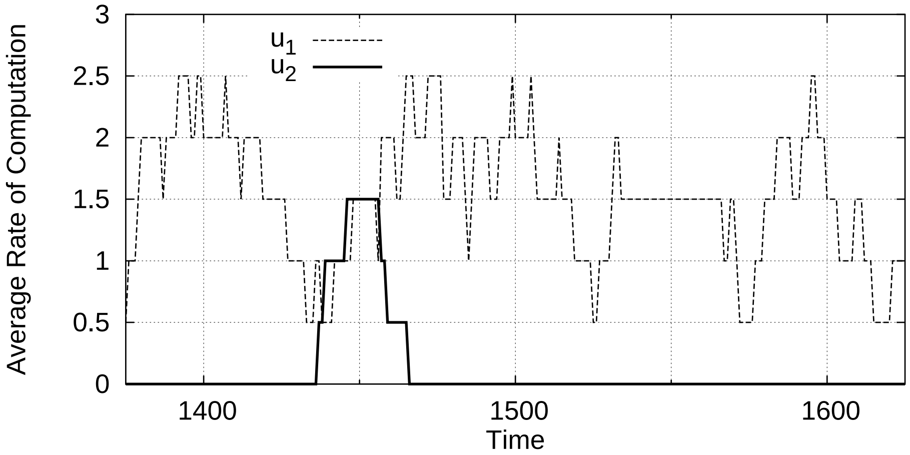


Figure 7.6. Graph of computational throughput for u_1 and u_2 over time

manager as the first. However, this user cheats by paying only the first 20% of the value of each contract. Figure 7.6 shows that the trust system protects against this behaviour without restricting user u_1's resource usage, in a graph which shows the computational throughput for both users over time as a running average of computations performed. This helps to validate the trust model's liveness and safety guarantees proved in Sections 4.2.2 and 4.2.3.

7.3 Compute Server Extensions and Limitations

There are a number of ways in which the compute server implementation of this section could be extended and enhanced, both to enable more varied contracts and to better support cooperation between federated servers in a computational Grid.

The first extension would be to allow native code execution and pre-emption in contracts. This can be achieved crudely with no operating system changes, by spawning native tasks as subprocesses of the contract and then using operating system calls to monitor their resource consumption periodically. Tasks which exceeded their resource budget or failed to make payments could also be temporarily suspended and resumed or killed outright through system calls. This extension would entail some change to the `ResourceSystem` class to allow resource monitoring and attribution of subprocesses. Similar changes could also be used for multi-CPU computers, in which multiple contracts can be executed simultaneously; the scheduler could again use the `ResourceSystem` to periodically authorise resources for contracts to use on other CPUs, and also to monitor ongoing resource usage. With a more controllable operating system kernel, the same mechanisms could be used for finer grained control.

The HTTP server, which currently performs its own resource measurements, could be modified to use the `ResourceSystem` interface to monitor resources. However, there are two limitations which would make this more difficult: firstly, the appropriate contract for a message may not be known in advance before significant resources are expended; secondly, the communication thread would not be able to update accounting values without introducing complex, cumbersome lock structures for each contract, or else it would need to continue to update the values indirectly — unlike the computation thread.

For extra safety, the HTTP server could be extended to provide greater protection against deliberate denial of service attack; firstly, messages from unknown sources could be refused at an early stage, and messages purporting to be from past collaborators could be only partly decoded initially, to ensure that they were indeed valid and properly encrypted. Secondly, the bar to entry could be raised by requiring a computational proof of work (such as HashCash) to be provided before exchanging keys. There are nevertheless limits to the protection that this approach can provide against a concerted attack; for example, if the compute server had a low bandwidth network link through a modem then that could be flooded with useless packets, regardless of the user's software protection.

The contract resource model can be adjusted to limit the number of contracts principals enter into simultaneously; this is achieved by assigning an extra loss of trust upon entering into a contract, with a corresponding trust gain when the contract is terminated. This could be extended to a distributed solution by having servers notify recommenders when their recommendations contributed to the signing of a contract, allowing them to adjust their trust values appropriately.

The current compute server implementation acts as an isolated server making its own contract decisions. For greater efficiency, such as when providing a Grid computing service, a number of servers could cooperate in accepting contracts by pooling information about their resource availability into a single place (in the reverse of the Contract Net [96] protocol). This node would then act as a broker for the others in deciding which contracts to accept. The broker could even be run as an independent service, acting as an auction-house and applying a mark-up on contracts it had outsourced to other nodes. Conversely, clients could pool their contract requests in a virtual market, which servers could monitor to establish suitable contracts.

This chapter has shown how the compute server framework can be used to implement a practical compute server which is able to identify profitable contracts through trust modelling. Support for recommendations allows trust bootstrapping, but still protects the server from attacks from its users while allowing profitable contracts to continue. This helps to validate the contract framework of Chapter 3 together with the trust model and safety and liveness proofs of Chapter 4. The implementation can also be extended into a general purpose Grid server, with support for native processes and for federation of servers using contract brokers. The following chapter concludes this dissertation, and outlines the relevance of the contract framework implementations for the framework as a whole.

Chapter 8

Conclusions

Computer systems need to control access to their resources, to manage sharing and to safely perform services for unknown principals. To do this effectively, they need an explicit resource model, and a way to assess others' actions in terms of it. This dissertation shows how the actions can be represented using resource-based computational contracts, together with a rich trust model which monitors and enforces contract compliance. These contracts establish a realistic virtual economy for producing autonomous yet accountable computer systems; autonomy can save users and administrators time and effort spent micro-managing computer systems, while accountability is important to ensure that the automatic decisions are appropriate.

Many computer applications face a shortage of resources and need an effective mechanism for monitoring and controlling their resource usage, both in isolated situations where many components must share the same resources, and when interacting with other trusted and untrusted principals for distributed tasks. Existing systems focus on individual aspects of this, but do not integrate well with each other because they have no common terms of reference.

The essential challenge is that computer systems have no higher level understanding of the effects of their actions, and therefore blindly perform the same actions in all contexts — such as a laptop that checks its hard disk for errors at the same time every day regardless of the state of the battery. This could be resolved with formal planning frameworks, but they provide a level of control that is too indirect for interactive computerised tasks. Instead, this dissertation proposes a resource model on an intermediate level, that can easily integrate traditional computational resources but is also expressive enough to represent more abstract concepts, ranging from resource generalisations (e.g.

'CPU time on any `ix86` server') and computerised actions (e.g. 'number of web pages downloaded') to external resources such as the user's time.

Contracts link computations to the resource model, by expressing agreements between clients and servers as promises to exchange resources, and binding them to the underlying computations that use the resources. Contracts also have an expressive accounting language for resource exchange that is computationally safe with predictable resource needs. This framework allows planning at a level that supports automatic cost-benefit analysis, but also corresponds directly to computational processes.

A trust model protects against unreliable or untrustworthy contract partners, by monitoring contractual promises against actual resource usage. This is important in open distributed applications to protect participants from attacks which steal or monopolise their resources, and in controlled applications for detecting and isolating failures. The contract framework's novel trust model assesses contract compliance in terms of resource usage and subjective cost functions. This general trust model applies to all implementations of the contract framework, and theoretical proofs show that it guarantees protection against resource loss without disrupting honest, faithfully performed contracts. These properties still hold even when the trust model is extended to support virtual communities, managed with personal trust recommendations.

Extensions of the resource model show that the contract framework can also support higher level applications, such as the sale of information online, with contracts that treat complex actions as primitive resources. However, contracts need to be able to monitor their performance; for some applications such as PDA collaboration, this means that contracts can take a supportive rôle so that the user provides extra information only for ambiguous decisions.

Finally, tests of two implementations of the contract framework for a distributed composite event service and a Grid compute server show experimentally that the contract framework is directly useful for practical applications, and lives up to its promises of resource protection.

The contract framework provides both a generic structure and an operational model for simultaneously performing and monitoring computer tasks. The resource abstraction allows efficient assessment and prediction of contracts' impact, but leaves scope for integrating external influences into the same model. Together with robust, distributed trust modelling, this enables self-enforcing, automatic contracts to rationally control resource usage in computer systems.

8.1 Further Work

The contract framework also gives rise to further extensions and applications: collaboration modelling, contract negotiation, contract nesting and transparent monitoring of web service applications.

Collaboration modelling schemes would allow contracts between more than two participants, such as between a client, a web server and a database server. These can currently be arranged as chains of pairwise contracts, but combined multi-party contracts would be more efficient and simpler to analyse. Collaboration modelling would also allow participants to explicitly arrange contracts for processing by any one of a group of servers.

Contract negotiation schemes should be analysed in more detail, to establish efficient contract transformations that participants can use to establish a mutually satisfactory contract.

Contract nesting would structure contract interrelations in a nested way, analogously to a transactional database access model. These nestings would also introduce dependencies between contracts, allowing parent contracts to take responsibility for their subcontracts.

Dispute resolution and testimonials of successful contract completion could be added to the contract framework. This would allow the use of evidence of incorrect contract performance, such as conflicting computational results from another source, for activing explicit penalty clauses in a contract, or for presentation to an external arbiter. Testimonials of past, successful contracts would also be needed, both as direct evidence for disputes, and as indirect evidence of trustworthiness.

Transparent monitoring of web services provides another application for the contract framework, to track the resource usage of web services per user and flag users needing excessive resources. This would be useful both for monitoring the deployment of services and protecting against attacks. The application would be able to assess the need for more servers, and identify the users and contracts contributing the most load. Giving control to the monitor would enable it to maintain good service for all users by throttling back service access appropriately for high load users, either automatically or after manual confirmation from the system administrator. The safety properties of the resource framework would also offer protection against attacks or broken clients of the web services. This monitoring and protecting would manage the load using the contract framework, to provide reliable and efficient service to the users.

These additions would extend the scope of the contract framework for greater efficiency and more commercial applications. This would further enhance its contribution as a general purpose tool for managing distributed computation that allows participants to take calculated risks and rationally decide on their actions, to produce autonomous yet accountable computer systems.

Bibliography

[1] Alfarez Abdul-Rahman and Stephen Hailes. Supporting trust in virtual communities. In *Hawaii International Conference on System Sciences 33*, pages 1769–1777, January 2000.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers— Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[3] N. Asokan, V. Shoup, and M. Waidner. Optimistic fair exchange of digital signatures. *IEEE Journal on Selected Areas in Communications*, 18(4):593–610, April 2000.

[4] J. L. Austin. *How to do Things with Words*. Oxford University Press, New York, 1962.

[5] Farag Azzedin and Muthucumaru Maheswaran. Integrating trust into grid resource management systems. In *International Conference on Parallel Processing (ICPP 2002)*, pages 47–54, August 2002.

[6] Adam Back. Hashcash — a denial of service counter-measure, August 2002. [Online]. Available: `http://www.hashcash.org/`.

[7] Jean Bacon, Ken Moody, and Walt Yao. A model of OASIS role-based access control and its support for active security. *ACM Transactions on Information and System Security*, 5(4):492–540, 2002.

[8] Boris Balacheff, Liqun Chen, David Plaquin, and Graeme Proudler. A trusted process to digitally sign a document. In *Proceedings of the 2001 workshop on new security paradigms*, pages 79–86. ACM Press, 2001.

[9] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, volume 1550 of *Lecture Notes in Computer Science*, pages 59–63. Springer-Verlag, April 1998.

[10] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings 1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[11] Mike Bond and Piotr Zieliński. Decimalisation table attacks for pin cracking. Technical Report UCAM-CL-TR-560, University of Cambridge Computer Laboratory, February 2003.

[12] Sviatoslav Brainov and Tuomas Sandholm. Contracting with an uncertain level of trust. In *Proceedings of the first ACM conference on Electronic commerce*, pages 15–21, Denver, CO, USA, November 1999.

[13] V. Breton, M. Medina, and J. Montagnat. DataGrid, prototype of a biomedical grid. *Methods of Information in Medicine*, 42(2):143–147, 2003.

[14] Rajkumar Buyya. *Economic-based Distributed Resource Management and Scheduling for Grid Computing.* PhD thesis, Monash University, Melbourne, Australia, April 2002.

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

[16] Marco Carbone, Mogens Nielsen, and Vladimiro Sassone. A formal model for trust in dynamic networks. Research Series RS-03-04, BRICS, Department of Computer Science, University of Aarhus, January 2003. EU Project SECURE IST-2001-32486 Deliverable 1.1.

[17] Luca Cardelli. Computation on wide area networks, 8,9,15,16 May 2001. Cambridge Minicourse Lectures. [Online]. Available: `http://www.luca.demon.co.uk/Slides.html`.

[18] C. Castelfranchi, R. Falcone, and F. de Rosis. Deceiving in GOLEM: How to strategically pilfer help. In *Autonomous Agent '98: Working notes of the Workshop on Deception, Fraud and Trust in Agent Societies*, 1998.

[19] Abhishek Chandra, Micah Adler, and Prashant Shenoy. Deadline fair scheduling: Bridging the theory and practice of proportionate fair scheduling in multiprocessor systems. In *Proceedings of the 7th IEEE Real-Time Technology and Applications Symposium (RTAS '01)*, pages 3–14, Taipei, Taiwan, June 2001.

[20] David Chaum, Amos Fiat, and Moni Naor. Untraceable electronic cash. In *Advances in Cryptology - CRYPTO '88, 8th Annual International Cryp-*

tology Conference, volume 403 of *Lecture Notes in Computer Science*, pages 319–327. Springer, 1990.

[21] Rita Chen and William Yeager. Poblano: A distributed trust model for peer-to-peer networks. Sun Microsystems, Inc. White Paper, 2001. [Online]. Available: `http://www.jxta.org/project/www/docs/trust.pdf`.

[22] Aspassia Daskalopulu, Theo Dimitrakos, and Tom Maibaum. E-contract fulfilment and agents' attitudes. In *ERCIM WG E-Commerce Workshop on The Role of Trust in e-Business*, Zurich, October 2001.

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

[24] Chrysanthos Dellarocas. Immunizing online reputation reporting systems against unfair ratings and discriminatory behavior. In *Proceedings of the 2nd ACM Conference on Electronic Commerce*, pages 150–157, Minneapolis, MN, USA, October 2000.

[25] Chrysanthos Dellarocas and Mark Klein. Designing robust, open electronic marketplaces of contract net agents. In *Proceedings of the 20th International Conference on Information Systems (ICIS)*, pages 495–500, Charlotte, NC, USA, December 1999.

[26] Roger Dingledine, Michael J. Freedman, and David Molnar. The Free Haven project: Distributed anonymous storage service. In *International Workshop on Design Issues in Anonymity and Unobservability*, volume 2009 of *Lecture Notes in Computer Science*, pages 67–95. Springer, 2001.

[27] Peter Sheridan Dodds, Roby Muhamad, and Duncan J. Watts. An experimental study of search in global social networks. *Science*, 301(5634):827–829, 8 August 2003.

[28] John R. Douceur. The Sybil attack. In *Proceedings of the IPTPS02 Workshop (LNCS 2429)*, pages 251–260, Cambridge, MA, USA, March 2002.

[29] Boris Dragovic, Evangelos Kotsovinos, Steven Hand, and Peter R. Pietzuch. XenoTrust: Event-Based Distributed Trust Management. In *Proceedings of Trust and Privacy in Digital Business (TrustBus'03). In conjunction with the 14th International Conference on Database and Expert Systems Applications (DEXA'03)*, Prague, Czech Republic, September 2003.

[30] Hector A. Duran-Limon and Gordon S. Blair. The importance of resource management in engineering distributed objects. In *Second International Workshop on Engineering Distributed Objects (EDO 2000)*, volume 1999 of *Lecture Notes in Computer Science*, pages 44–60. Springer, November 2001.

[31] D. Eastlake, III. Cybercash credit card protocol. RFC 1898, Internet Engineering Task Force, February 1996. [Online]. Available: `http://www.ietf.org/rfc/rfc1898.txt`.

[32] D. Eastlake, III. US secure hash algorithm 1 (SHA1). RFC 3174, Internet Engineering Task Force, September 2001. [Online]. Available: `http://www.ietf.org/rfc/rfc3174.txt`.

[33] Richard J. Edell and Pravin P. Varaiya. Providing internet access: What we learn from INDEX. *IEEE Network*, 13(5):18–25, Sept–Oct 1999.

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

[35] Tzilla Elrad, Mehmet Aksits, Gregor Kiczales, Karl Lieberherr, and Harold Ossher. Discussing aspects of AOP. *Communications of the ACM*, 44(10):33–38, 2001.

[36] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[37] Dror G. Feitelson and Larry Rudolph. Metrics and benchmarking for parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing (IPPS/SPDP'98)*, volume 1459 of *Lecture Notes in Computer Science*, pages 1–24. Springer, 1998.

[38] Ian Foster and Carl Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[39] Juan A. Garay, Markus Jakobsson, and Philip MacKenzie. Abuse-free optimistic contract signing. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference*, volume 1666 of *Lecture Notes in Computer Science*, pages 449–466. Springer, 1999.

[40] David Garlan, Dan Siewiorek, Asim Smailagic, and Peter Steenkiste. Project Aura: Towards distraction-free pervasive computing. *IEEE Pervasive Computing*, 1(2):22–31, 2002.

[41] Kurt Geihs. Adaptation strategies for applications in dynamic environments. In *Middleware for Mobile Computing Workshop, In association with IFIP/ACM Middleware 2001 Conference*, 2001.

[42] Al Geist, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidyalingam S. Sunderam. *PVM: Parallel Virtual Machine: A Users' Guide and Tutorial for Networked Parallel Computing*. Scientific and engineering computation. MIT Press, Cambridge, MA, USA, 1994.

[43] David Gibson. Analysing reputation in web communities. In *International Conference on Internet Computing 2003*, pages 10–16, 2003.

[44] Li Gong, Marianne Mueller, Hemma Prafullchandra, and Roland 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*, Monterey, CA, USA, December 1997.

[45] Elizabeth Gray, Paul O'Connell, Christian Jensen, Stefan Weber, Jean-Marc Seigneur, and Chen Yong. Towards a framework for assessing trust-based admission control in collaborative ad hoc applications. Technical Report TCD-CS-2002-66, Trinity College Dublin, 2002.

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

[47] Martin Hirt, Ueli M. Maurer, and Bartosz Przydatek. Efficient secure multi-party computation. In *ASIACRYPT 2000*, pages 143–161, 2000.

[48] R. Housley, W. Ford, W. Polk, and D. Solo. Internet X.509 public key infrastructure certificate and CRL profile. RFC 2459, Internet Engineering Task Force, January 1999. [Online]. Available: `http://www.ietf.org/rfc/rfc2459.txt`.

[49] iLumin Corporation. Introduction to the automated enforceable online transactions market, 2000. [Online]. Available: `http://www.ilumin.com/dhs/AEOLT_Whitepaper.pdf` [16 May 2001].

[50] David Ingram. Trust-based filtering for augmented reality. In *Proceedings of the First International Conference on Trust Management (iTrust*

2003), volume 2692 of *Lecture Notes in Computer Science*, pages 108–122. Springer, May 2003.

[51] Ari K. Jonsson, Paul H. Morris, Nicola Muscettola, Kanna Rajan, and Benjamin D. Smith. Planning in interplanetary space: Theory and practice. In *Artificial Intelligence Planning Systems*, pages 177–186, 2000.

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

[53] Radu Jurca and Boi Faltings. An incentive compatible reputation mechanism. In *IEEE International Conference on E-Commerce (CEC 2003)*, pages 285–292, Newport Beach, CA, USA, June 2003.

[54] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Héctor M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Janotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Symposium on Operating Systems Principles*, pages 52–65, 1997.

[55] Anne-Marie Kermarrec, Laurent Massouli, and Ayalvadi J. Ganesh. Probabilistic reliable dissemination in large-scale systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(3):248–258, 2003.

[56] Jörg Kienzle and Rachid Guerraoui. AOP: Does it make sense? The case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP 2002)*, volume 2374 of *Lecture Notes in Computer Science*, pages 37–61. Springer, 2002.

[57] Mark Klein, Peyman Faratin, Hiroki Sayama, and Yaneer Bar-Yam. Protocols for negotiating complex contracts. *IEEE Intelligent Systems*, 18(6):32–38, 2003.

[58] Fabio Kon, Fabio Costa, Gordon Blair, and Roy H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[59] Eric Korpela, Dan Werthimer, David Anderson, Jeff Cobb, and Matt Lebofsky. SETI@home—massively distributed computing for SETI. *Computing in Science and Engineering*, 3(1):78–83, January 2001. Available: `http://www.computer.org/cise/articles/seti.htm` [15 Oct 2003].

[60] Steven T. Kuhn. Prisoner's dilemma. In Edward 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/`.

[61] Yannis Labrou and Tim Finin. A proposal for a new KQML specification. Technical Report TR CS-97-03, Computer Science and Electrical Engineering Department, University of Maryland Baltimore County, Baltimore, MD 21250, USA, February 1997. [Online]. Available: `http://www.cs.umbc.edu/kqml/`.

[62] George Lakoff. *Women, Fire, and Dangerous Things. What Categories Reveal About the Mind*. Chicago: University of Chicago Press, 1987.

[63] Pradip Lamsal. Understanding trust and security, October 2001. [Online]. Available: `http://www.cs.helsinki.fi/u/lamsal/papers/UnderstandingTrustAndSecurity.pdf` [15 Dec 2002].

[64] Marc Langheinrich. When trust does not compute - the role of trust in ubiquitous computing. Workshop on Privacy at Ubicomp 2003, Seattle, WA, USA, October 2003.

[65] J. Larmouth. Scheduling for a share of the machine. Technical Report 2, University of Cambridge Computer Laboratory, 1974.

[66] Victor R. Lesser. Cooperative multiagent systems: A personal view of the state of the art. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):133–142, Jan–Feb 1999.

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

[68] Leonidas Lymberopoulos, Emil Lupu, and Morris Sloman. An adaptive policy based management framework for differentiated services networks. In *Policy 2002: IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 147–158, Monterey, CA, USA, June 2002.

[69] Olivier Markowitch, Dieter Gollmann, and Steve Kremer. On fairness in exchange protocols. In *Information Security and Cryptology (ICISC 2002)*, volume 2587 of *Lecture Notes in Computer Science*, pages 451–464. Springer, 2002.

[70] S. Marsh. *Formalising Trust as a Computational Concept.* PhD thesis, University of Stirling, Department of Computer Science and Mathematics, UK, 1994.

[71] D. Harrison McKnight and Norman L. Chervany. What is trust? A conceptual analysis and an interdisciplinary model. In *Americas Conference on Information Systems (AIS 2000).* Omnipress, Long Beach, CA, 2000.

[72] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography.* CRC Press, Boca Raton, FL, USA, 1996.

[73] Mark S. Miller. Partial trust, August 1997. [Online]. Available: `http://discuss.foresight.org/critmail/webenh-content/0081.html` [19 Jul 2001].

[74] Aloysius K. Mok and Weijiang Yu. TINMAN: A resource bound security checking system for mobile code. In *7th European Symposium on Research in Computer Security Computer Security (ESORICS 2002)*, volume 2502 of *Lecture Notes in Computer Science*, pages 178–193, Zurich, Switzerland, October 2002. Springer.

[75] Carlos Molina-Jimenez and John Warne. TAPAS architecture: concepts and protocols, April 2002. EU Project TAPAS IST-2001-34069 Deliverable D5. [Online]. Available: `http://www.newcastle.research.ec.org/tapas/deliverables/` [15 Oct 2003].

[76] David Mosberger. A closer look at the Linux $O(1)$ scheduler, 2003. [Online]. Available: `http://www.hpl.hp.com/research/linux/kernel/o1.php` [15 Oct 2003].

[77] George C. Necula. Proof-carrying code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[78] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, 1998. [Online]. Available `http://dbpubs.stanford.edu:8090/pub/1999-66`.

[79] Jens Palsberg and Peter Ørbæk. Trust in the λ-calculus. Research Series RS-95-31, BRICS, Department of Computer Science, University of Aarhus, Aarhus, Denmark, June 1995. Appears in *Proceedings of 2nd International Static Analysis Symposium (SAS'95)*, 1995, pages 314–330.

[80] Christos H. Papadimitriou. *Computational Complexity.* Addison-Wesley, New York, 1994.

[81] Martin Pelikan, David Goldberg, and Fernando Lobo. A survey of optimization by building and using probabilistic models. Technical Report 99018, Illinois Genetic Algorithms Laboratory, University of Illinois at Urbana-Champaign, IL, USA, September 1999.

[82] Peter R Pietzuch, Brian Shand, and Jean Bacon. A framework for event composition in distributed systems. In *4th International Conference on Middleware (Middleware'03)*, pages 62–82, Rio de Janeiro, Brazil, June 2003.

[83] PlanetLab. An open platform for developing, deploying, and accessing planetary-scale services, 2003. [Online]. Available: `http://www.planet-lab.org/` [15 Oct 2003].

[84] Nuno Preguiça, Marc Shapiro, and Caroline Matheson. Efficient semantics-aware reconciliation for optimistic write sharing. Technical Report MSR-TR-2002-52, Microsoft Research, May 2002.

[85] Dickon Reed, Ian Pratt, Paul Menage, Stephen Early, and Neil Stratford. Xenoservers: accountable execution of untrusted code. In *IEEE Hot Topics in Operating Systems (HotOS) VII*, March 1999.

[86] Susanne Rhrig. Using process models to analyze health care security requirements. In *International Conference on Advances in Infrastructure for e-Business, e-Education, e-Science, and e-Medicine on the Internet*, L'Aquila, Italy, 2002.

[87] Timothy Roscoe. The structure of a multi-service operating system. Technical Report UCAM-CL-TR-376, University of Cambridge Computer Laboratory, August 1995. (PhD thesis).

[88] Tuomas Sandholm and Victor Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *First International Conference on Multiagent Systems (ICMAS-95)*, pages 328–335, San Francisco, CA, USA, 1995.

[89] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[90] M. Angela Sasse. Computer security: Anatomy of a usability disaster, and a plan for recovery. In *CHI 2003 Workshop on Human-Computer Interaction and Security Systems*, Ft. Lauderdale, FL, USA, April 2003.

[91] Jean-Marc Seigneur, S. Farrell, Christian Jensen, Elizabeth Gray, and Chen Yong. End-to-end Trust Starts with Recognition. In *Security in Pervasive Computing, First International Conference*, volume 2802 of *Lecture Notes in Computer Science*, pages 130–142. Springer, March 2003.

[92] Peter Sewell and Jan Vitek. Secure composition of untrusted code: Wrappers and causality types. In *Proceedings of The 13th Computer Security Foundations Workshop (CSFW'00)*, pages 269–284. IEEE Computer Society Press, 2000.

[93] Brian Shand and Jean Bacon. Policies in accountable contracts. In *Policy 2002: IEEE 3rd International Workshop on Policies for Distributed Systems and Networks*, pages 80–91, Monterey, CA, USA, June 2002.

[94] Brian Shand, Nathan Dimmock, and Jean Bacon. Trust for transparent, ubiquitous collaboration. In *First IEEE Annual Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 153–160, Dallas-Ft. Worth, TX, USA, March 2003.

[95] Abraham Silberschatz and Peter Baer Galvin. *Operating System Concepts*. Addison-Wesley, Reading, MA, USA, 1998.

[96] Reid G. Smith. The contract net protocol: High-level communication and control in a distributed problem solver. In Alan H. Bond and Les Gasser, editors, *Readings in Distributed Artificial Intelligence*, chapter 5.2, pages 357–366. Morgan Kaufmann Publishers, 1988.

[97] Sun. Java$^{\text{TM}}$ Message Service. `http://java.sun.com/products/jms/`, 2001.

[98] SunSpot.net. Baltimore man pleads guilty in bogus eBay sale of Ferrari, June 2003. [Online]. Available: `http://web.archive.org/web/20030620143047/http://www.sunspot.net/news/local/bal-ferrari0617,0,1455494.story` [15 Oct 2003].

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

[100] Sameer Tilak, Nael B. Abu-Ghazaleh, and Wendi Heinzelman. A taxonomy of wireless micro-sensor network models. *ACM Mobile Computing and Communications Review (MC^{2}R)*, 6(2):28–36, April 2002.

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

[102] Ian Wakeman, David Ellis, Tim Owen, Julian Rathke, and Des Watson. Risky business: Motivations for markets in programmable networks. In *International Workshop on Active Networks (IWAN)*, Kyoto, Japan, December 2003.

[103] Carl A. Waldspurger, Tad Hogg, Bernardo A. Huberman, Jeffrey O. Kephart, and W. Scott Stornetta. Spawn: A distributed computational economy. *IEEE Transactions on Software Engineering*, 18(2):103–117, February 1992.

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

[105] Nanbor Wang, Douglas C. Schmidt, Michael Kircher, and Kirthika Parameswaran. Adaptive and reflective middleware for QoS-enabled CCM applications. *IEEE Distributed Systems Online*, 2(5), 2001. [Online]. Available: `http://dsonline.computer.org/0105`.

[106] Yao Wang and J. Vassileva. Bayesian network-based trust model. In *IEEE/WIC International Conference on Web Intelligence (WI 2003)*, pages 372–378, Halifax, Canada, October 2003.

[107] Stephen Weeks. Understanding trust management systems. In *IEEE Symposium on Security and Privacy*, pages 94–105, 2001.

[108] Michael P. Wellman and Peter R. Wurman. Market-aware agents for a multiagent world. *Robotics and Autonomous Systems*, 24(3):115–125, September 1998.

[109] Bryce Wilcox-O'Hearn. Experiences deploying a large-scale emergent network. In *Peer-to-Peer Systems, First International Workshop, IPTPS 2002*, volume 2429 of *Lecture Notes in Computer Science*, pages 104–110. Springer, 2002.

[110] John Wilkes, Patrick Goldsack, G. (John) Janakiraman, Lance Russell, Sharad Singhal, and Andrew Thomas. eOS — the dawn of the resource

economy. Technical report, HP Laboratories, Palo Alto, CA, USA, May 2001.

[111] Fredrik Ygge and Hans Akkermans. Making a case for multi-agent systems. In *8th European Workshop on Modelling Autonomous Agents in a Multi-Agent World (MAAMAW)*, volume 1237 of *Lecture Notes in Computer Science*, pages 156–176. Springer, 1997. [Online]. Available: `http://www.enersearch.se/ygge`.