The Dissertation Committee for Kevin Michael Kane
certifies that this is the approved version of the following dissertation:

Access Control in Decentralized, Distributed Systems

Committee:

James C. Browne, Supervisor

Lorenzo Alvisi

Michael D. Dahlin

Vijay K. Garg

Mohamed G. Gouda

Vitaly Shmatikov

Access Control in Decentralized, Distributed Systems

by

Kevin Michael Kane, M.S., B.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 2006

Acknowledgments

I would like to thank many people who have helped me during my time in graduate school. First and foremost I must thank my advisor, Professor James Browne, for taking me in as a new graduate student, enduring our struggles to find a common area of research between his interests and mine, and helping bring me from being a student who knew only absorbing information from coursework and textbooks to being a research professional participating in an active community of scholars, contributing as well as absorbing. He has shown me his insatiable hunger to both consume and produce new knowledge. Jim Browne is a teacher and researcher who is not happy unless he is reading a paper, working on research, or writing a paper on that research – sometimes simultaneously!

I am also grateful to my doctoral committee, Professors Lorenzo Alvisi, Michael Dahlin, Vijay Garg, Mohammed Gouda, and Vitaly Shmatikov, for seeing enough substance in my rocky dissertation proposal to allow me to proceed, but insisting that I turn my rocky start into a quality piece of work before I would receive their approval. That their signatures now grace the certification page shows how their insistence to excellence in their students taught me how to do good work.

When I visited for the department's GradFest in 2000 I met and spoke with Professor Calvin Lin. My application to graduate school was still under consideration at the time. He was kind enough to keep in contact with me during my application's consideration, and when I was accepted, still kept in contact while

funding was found for me. I am grateful for his interest and help during that time.

I also must thank Allen Lambert, who helped me through a particularly difficult time in not only my research but my life in general, and taught me that above all else, my first priority must always be to take care of myself. Further, he taught me that there is no point in worrying about possible problems: by their very definition, a possible problem is one that cannot be solved for it does not yet exist, and so there is no point in worrying about it. He taught me instead to focus on the real challenges before me, because they can be addressed, solved, and put behind me. "Don't borrow trouble," he advised, and I have done my best to follow that advice.

My first mentor while at Maryland was also the instructor of the C++ course I took my first semester, Gwen Kaye. The new C++ curriculum had just been implemented that semester, and I ended up getting to know Gwen rather well by spotting problems with the projects and working with her to correct them. As the undergraduate program coordinator and one of the academic advisors, I would make a point to see her specifically for the required advising before registration each semester. She encouraged me to get involved in the undergraduate teaching assistant program where I served for most of my undergraduate career. When I was honored as an outstanding graduating senior, I did not hesitate to share the moment with her as my mentor.

Thanks also to Dr. Michelle Hugue, one of the most extraordinary teachers it has been my pleasure to study under, who I came to know as an instructor at the University of Maryland and later as a friend. She is partly to blame for my going all this way. I originally intended only to pursue my Master's degree, but she convinced me to apply to doctoral programs just to maximize my opportunities in a devious ploy to make me a researcher. Before I knew it, I was involved in research and working for my doctorate.

The support and advice of former and current graduate students was also helpful and encouraging, particularly at times when my future as a doctoral student seemed uncertain. I would like to thank, in no particular order, the following people: Fei Xie, Huaiyu (Kitty) Liu, Nasim Mahmood, Maria Jump, Jean-Philippe Martin, Alison Norman, and Kartik Agaram. Their advice and assurance that my experiences were normal encouraged me to carry on. Maria in particular, who I knew at Maryland, even put me up for a couple days when I first arrived in Austin and was very helpful in getting me settled into my new home.

I would also like to thank Dr. Tommy Darwin, whose course in dissertation and grant writing I had the pleasure of taking, who offered valuable advice on the writing of scholarly works, and showed me that there is more to writing papers, reports, and dissertations than being a repository of a body of knowledge. They do not exist in isolation. They are living documents that will try to persuade an audience that the ideas they contain are valid and meaningful, and must be crafted specifically to that task, and customized to that audience. He showed me there is a lot more to writing a paper than listing results, and that there are a lot of useful techniques that can be used to turn a good paper into a great paper.

A rather unusual thanks to Don Geronimo and Mike O'Meara, the hosts of a radio program originating in my native Washington, DC which I can now hear in Austin, thanks to the Internet. I have had a lot of stressful days and nights, where the stress would be so heavy it would make it impossible to work or even sleep. Listening to their comedy program, both live and recorded, never failed to untie the knots in my stomach tied by the pressures of my work.

This work has been supported in part by the State of Texas Coordinating Board for Higher Education under TARP grant 003658-0508-1999, the National Science Foundation under grant numbers 0103725, "Performance-Driven Adaptive Software Design and Control" and ACI-0305644, "Montage: An Integrated End-

to-End Design and Development Framework for Wireless Networks," and by the Defense Advanced Research Projects Agency (DARPA) under contract number NBCH30390004.

Kevin Michael Kane

The University of Texas at Austin
December 2006

Access Control in Decentralized, Distributed Systems

Publication No. _____

Kevin Michael Kane, Ph.D.
The University of Texas at Austin, 2006

Supervisor: James C. Browne

Distributed systems with decentralized control, such as peer-to-peer systems, computing grids across multiple organizations, and compositional web services require a rethinking of basic issues in their design and implementation. This dissertation establishes a model of these systems, examines the issues of programming models, access control, and trust, and proposes and evaluates new methods for implementing access control and trust management in these systems.

We begin by abstracting these systems into the model of *service-oriented systems* that use associative interactions. It then examines the fundamental issues of access control in decentralized systems, and its greater requirements than in systems with central control. From this analysis we identify possible solutions for access con-

trol implementations that have not been explored. We then introduce a framework developed to support computation in these kinds of networks, and then together with the lessons learned from the taxonomy, offer and evaluate *contractually-limited capabilities* as an access control mechanism.

We then address the questions of trust, cooperation, and access control decisions by offering and evaluating a reputation-tracking mechanism that incorporates a quantified measure of *uncertainty*, realizing that intrinsic to any system with decentralized control is the uncertainty of information arising from incomplete state. We show this mechanism promotes cooperation by throttling uncooperative nodes while providing high levels of service to cooperative nodes. We then further examine the question of reliability of these networks by introducing a logic for verifying access control properties. This logic unifies a logic of belief with temporal logic, and establishes formal models of these systems that can then be proven to possess desirable properties.

Contents

Acknowledgments **iv**

Abstract **viii**

List of Tables **xv**

List of Figures **xvi**

Chapter 1 Introduction **1**

1.1 Problem Statement 1

1.2 Contributions of this Dissertation 2

1.3 Layout of this Dissertation 3

Chapter 2 Decentralized, Distributed Service-Oriented Systems **6**

2.1 Model Overview 6

2.1.1 Definitions 6

2.2 Associative Broadcast 9

2.3 Related Work 11

2.4 Conclusions 11

Chapter 3 A Taxonomy of Access Control **13**

3.1 Introduction 13

3.2 Related Work 14

3.2.1 Surveys and Descriptions 15

3.2.2 Comparison of Expressiveness of Schemes and Models 15

3.3 Definitions 16

3.3.1 Access Control Policies, Schemes, and Implementations . . . 16
3.3.2 Implementations versus Schemes . . . 17
3.3.3 Access Control Credential . . . 17
3.3.4 Lattice Taxonomy . . . 18
3.4 Classification Axes . . . 18
3.4.1 Control over Sharing of Access Credentials . . . 18
3.4.2 State Distribution . . . 20
3.4.3 Fidelity of Enforcement . . . 21
3.4.4 Identity Resolution . . . 23
3.4.5 Decision Mode . . . 24
3.4.6 Static or Adaptive Trust Management . . . 25
3.5 Example Classifications . . . 26
3.5.1 Akenti . . . 26
3.5.2 CRISIS . . . 27
3.5.3 dRBAC . . . 28
3.5.4 Kraft-Schäfer Mobile Ad-Hoc Networks . . . 29
3.5.5 BitTorrent . . . 31
3.5.6 Wi-Fi Protected Access in IEEE 802.11 Wireless Networks . 32
3.6 Evaluation . . . 34
3.7 Conclusions . . . 36

Chapter 4 CoorSet: A Development Environment for Associatively Coordinated Components 37

4.1 Introduction . . . 37
4.2 Broadcast-Based Coordination . . . 38
4.3 Associative Interactions . . . 39
4.3.1 Coordination and Composition under Associative Broadcast . 39
4.3.2 Composition as Coordination and Vice Versa . . . 40
4.4 Coordination and Composition . . . 41
4.4.1 Naming Models and Communication Models . . . 41
4.4.2 A Coordination-Oriented Implementation of Associative Interactions . . . 41
4.5 Extended Associative Broadcast Coordination and Programming Model 42
4.6 Algorithm Formulation . . . 44

4.7 CoorSet Interface Definition Language 44
4.7.1 Readers/Writers Algorithm in CoorSet 45
4.7.2 Data Fitting Example 47
4.7.3 Distributed Computation of Google PageRanks 50
4.8 Runtime Support for Launching 52
4.8.1 Component Starting Component 53
4.8.2 Grid MP 54
4.9 Implementation 56
4.10 Related Research 57
4.10.1 Associative Broadcast 57
4.10.2 Runtime Systems 58
4.10.3 Coordination Languages 59
4.11 Conclusions 61

Chapter 5 Contractually-Limited Capabilities and CapCoorSet 62
5.1 Introduction 62
5.2 Capability-Based Access Control 63
5.2.1 System Architecture 64
5.2.2 Fundamentals of Capabilities 65
5.2.3 Contractually-Limited Capabilities 66
5.2.4 CapCoorSet 67
5.3 Properties 69
5.4 Example: Stock Trading 71
5.5 Language, Implementation, and Evaluation 75
5.6 Related Work 77
5.7 Conclusions 78

Chapter 6 Automated Trust Decisions using Subjective Logic 79
6.1 Introduction 79
6.1.1 System Model 81
6.2 Related Work 82
6.3 Reputations with Uncertainty 84
6.3.1 Subjective Logic 84
6.3.2 Forming Opinions 84

6.3.3 Using Cooperation to Overcome Uncertainty 86
6.3.4 Protocol Behavior 88
6.4 Simulation Architecture 89
6.5 Performance Analysis 91
6.5.1 Metrics 91
6.5.2 Simulation Results 92
6.6 Conclusions 96

Chapter 7 Logic for Verification of Access Control Properties 98
7.1 Introduction 98
7.2 Related Work 100
7.3 Syntax 101
7.3.1 Informal Description of Syntax 101
7.4 Semantics 103
7.5 Example Formulas 108
7.6 Example Proofs 111
7.7 Conclusions 114

Chapter 8 Conclusions and Future Work 115
8.1 Future Research Directions 116
8.1.1 Taxonomy of Access Control 116
8.1.2 Contractually-Limited Capabilities and CapCoorSet 117
8.1.3 Automated Trust Decisions 117
8.1.4 Verification of Service-Oriented Systems 119

Appendix A CoorSet Interface Description Language 120
A.1 Building and Running Applications 120
A.1.1 Running the Compiler 120
A.1.2 Starting Your Application 121
A.1.3 Network Parameters 122
A.1.4 Component Types 124
A.2 Runtime System, API Description, and Current Status 129
A.2.1 Supporting Classes Common to All Types 130
A.2.2 Self-Contained Component Interface 133
A.2.3 Firing Rule Interface 134

A.2.4 Complex Component Interface 135
A.3 The Component Starting Component (CSC) 137
A.3.1 The Key Generator 137
A.3.2 The Component Starting Component 138
A.4 Directory Server for United Devices 138

Appendix B CapCoorSet **139**
B.1 Condensed Broker Definition 139
B.2 Stock Example Workflow 140

Appendix C Verification Logic Soundness Proof **144**

Bibliography **148**

Vita **162**

List of Tables

5.1 Customer Requires Interface 72
5.2 Stock Broker Accepts Interface 72
5.3 Stock Broker Requires Interface 73
5.4 Stock Issuer Accepts Interface 73
5.5 Bank Accepts Interface 74

6.1 Run length to total detection in a network of 40 services 94

List of Figures

4.1 CoorSet definition of a replicated data object store 45
4.2 Data flow between components of the data fitter application 48
4.3 Initial configuration of data fitting in *CoorSet* 51

6.1 Belief/disbelief/uncertainty triangle 85

7.1 Syntax of the Service Logic 101

B.1 Broker condensed definition in CapCoorSet language 140
B.2 Stock Market Example 141

Chapter 1

Introduction

1.1 Problem Statement

The rise of Internet-based systems and systems with distributed control, such as peer-to-peer systems [38, 107, 132] and compositional web services [7, 82, 129], requires a rethinking of basic issues ranging from programming models to access control. These systems must function in the presence of uncertain and incomplete system state, and the presence of selfish or malicious agents. Programming models based on central control and complete system state are no longer applicable. Components of these systems that work on Internet scale must be based on cooperation and coordination. These must be accomplished through programming models and implementation mechanisms that tolerate asynchrony, and do not rely on assumptions of static network membership, static trust relationships between components, or perfect and complete state information. Components in these systems concerned for their self-interest must act both to reach out for cooperation to accomplish shared goals while protecting local resources from abuse or compromise. This dissertation revisits the problems of access control and trust in distributed, decentralized systems.

Traditional distributed systems benefit from the certainty of a static arrangement and division of work, and established trust relationships through a known and unquestioned authentication and authorization hierarchy. Programming models can safely assume a static and known population of services that obey a known protocol and will only make requests or take actions that support the system's common goal.

Reliability of services is assured. Identity certification and access control decisions are all made by recognized and unequivocally trusted authorities.

Distributed systems that employ decentralized control are under the control of more than one administrative entity. These systems include computing grids spanning multiple virtual organizations [54], peer-to-peer networks, mobile ad-hoc networks, and components of a compound web service. In the most extreme case, each individual node is separately controlled, and while there may be global agreement on interaction protocols and desired goals, there is no agreed-upon authority to certify a node's identity, no pre-existing trust relationships, and no guarantee all nodes will contribute to the established goals. These networks can grow to arbitrary size, which can make keeping track of all nodes and their authorizations difficult, if it is even possible. New approaches are required for access control in these systems.

1.2 Contributions of this Dissertation

To provide a basis for designing and analyzing these new approaches, this dissertation first contributes a taxonomy of access control implementations [75] that decomposes the space into a set of axes, each representing a design choice when creating an implementation. Central to the functionality of decentralized systems is the method of access control. Although historically access control has been the method of authenticating users and allowing a known population of known users to access required resources, in decentralized systems access control becomes important to the overall function of the network, and not just defending it from outside adversaries. Access control becomes a necessary tool in both providing the ability to cooperate, as well as providing incentive and enforcement to ensure that nodes cooperate. There has not been a careful examination of the tradeoffs and issues involved with design choices made when choosing an access control implementation. Several distributed systems with access control are analyzed and classified relative to this taxonomy. This exposes unoccupied space in the taxonomy.

This unoccupied space motivates the development of a framework for computation in decentralized systems with a standard mechanism for access control called CapCoorSet. This is the second contribution of this dissertation. CapCoorSet occupies one of the unexplored areas identified by the taxonomy: an implementation that allows delegation of authority in a controlled manner, and stores most of the

access control-related state on the client side.

CapCoorSet leaves to the application programmer how to make the access control decision. Although simple, statically-programmed methods are possible, the nature of decentralized systems requires a method that can cope with changing conditions of the network and changing behaviors of other nodes. The third contribution of this work is a method of tracking reputation for other nodes [76] that observes the history of cooperation or lack thereof from other nodes and uses it as the access control decision when asked to cooperate itself. This method not only rewards cooperation with cooperation, but also recognizes that its information is incomplete and therefore uncertain, and accounts for that in its computations.

It can be shown through testing and experimentation that these systems function well in practice, but no amount of testing can exhaustively explore every possible execution of a system. The only means to guarantee properties of such a system is through formal verification. The question of verification in general has been well-explored, but none have yet been customized to these types of systems. To answer this, the fourth contribution of this dissertation is a logic for service-oriented systems, which provides a formalism in which to easily describe properties of these systems, and mechanisms in which to prove them.

1.3 Layout of this Dissertation

Chapter 2 introduces the service-oriented model of decentralized systems, including its method of interaction, and capabilities, the mechanism that will later be used for access control. Chapter 3 gives a taxonomy of access control that examines the space of possible access control implementations in these systems. Implementations are often categorized by the abstraction used to design the policy, such as user-based or role-based, or by the mechanism used for the decision, such as access control lists or capabilities, but these do not distinguish between centralized and distributed implementations, give insight into the relative robustness of different implementations, or define the space of possible implementations. We analyze a number of current implementations and classify them in the taxonomy, and as a result identify a number of unexplored design choices.

As a basis for evaluating these unexplored design spaces, chapter 4 presents CoorSet [74], which stands for Coordinated Set, a framework for computation in this

decentralized setting. It uses associative interactions as its model of interaction, with well-defined interfaces and dynamic binding of composed services to tolerate asynchronous and unreliable interactions. This chapter describes the implementation of the CoorSet system, and applications built on top of it.

CoorSet itself provides only the model of interaction and a supporting runtime system, but no access control on the services themselves, leaving the application programmer to provide any desired access control. The unoccupied space identified in the taxonomy in chapter 3 motivates the addition of a standard access control mechanism to CoorSet. The taxonomy shows that access control implementations that allow the clients some control over delegating their authority have not been well-explored. It also shows that implementations that distribute the storage of access control-related state to minimize the cost on the services also have not been well-explored. This improves that system's scalability in the face of a large number of clients. Chapter 5 uses this set of design choices to provide access control for operations provided by nodes in a CoorSet system, which is already designed for computation in decentralized systems. Access control for operations is provided using an extended form of capabilities called *contractually-limited capabilities* (CLCs). Each capability has an associated contract that specifies conditions on the capability's use. The application of CLCs to CoorSet is called CapCoorSet, for capability-protected CoorSet. The contracts can limit the propagation of capabilities, which is a shortcoming of unrestricted capabilities [26]. The CLC is stored by the client, and submitted to the service to make a request. This solution works well to allow some control over delegation to be given to the client, while leaving enough with the service to avoid the problems of unbounded propagation of capabilities.

The application decides to grant access to an operation by issuing a CLC to a client with an appropriate contract, or refusing to do so. The simplest choice for this decision is to program an access control list that can be checked when a node requests access to a service. Although this is functional and acceptable, it does not take advantage of the ability to distribute access control-related state, as the access control list must be stored at the service. Additionally, in systems with increasingly decentralized control such as those found in peer-to-peer and ad-hoc systems, it is unlikely that a node will know a list of other nodes with which it should cooperate *a priori*. Worse, nodes can change their behavior and become untrustworthy, making such a static approach undesirable.

The taxonomy shows a lack of implementations where nodes automatically monitor the behavior of interaction partners, and implementations that make use of information from other nodes in formulating their decisions; typically decisions are made either by a central authority where one exists, or entirely locally by the node where it does not. Solutions using a combination of decision-makers have only recently been explored [72, 78, 31]. Chapter 6 introduces a new approach to reputation-based automated trust management that not only tracks the behavior of other nodes as an indicator of trustworthiness, but also tracks how uncertain that information is. A reputation-based rating may be uncertain due to a node being unknown, or altering its behavior over time. This approach supports polling other nodes with more information for recommendations, when the local uncertainty is too high. This allows nodes to benefit from a history of cooperation with other nodes, to allow them to more rapidly receive service with new interaction partners, and to allow nodes that have acted uncooperatively to more rapidly be detected and throttled. We show experimental results that show our method is competitive with other approaches when the uncooperative population is in the minority, and provides superior performance in many cases with regards to the time required to detect uncooperative behavior and the ability to limit the resource consumption of uncooperative nodes, while ensuring that cooperative nodes continue to be able to function. This mechanism can be integrated into the infrastructure of CapCoorSet, or applied to other systems where dynamic trust management is required.

Although we can show experimentally that systems using these techniques function well, we desire a method to prove properties about these systems. Chapter 7 describes a logic for proving access control properties about service-oriented distributed systems. We give the syntax and semantics of the logic, followed by a set of example statements and proofs in the logic demonstrating its use. Chapter 8 sums up the contributions of this work and examines its impact on the field of decentralized, distributed systems, and explores a variety of future avenues of research in the various problem domains explored by this dissertation.

Because this dissertation explores several different aspects of access control in decentralized systems, we explore related work in each area in its respective chapter. Therefore in the next chapter, we begin with the exploration of the space of access control implementations to provide a greater understanding of the trade-offs when selecting an implementation, and also to motivate the later work in this dissertation.

Chapter 2

Decentralized, Distributed Service-Oriented Systems

We abstract grid- or Internet-based systems into the model of a *service-oriented system*. We call each node a *service* to emphasize its active role in its access control. This is in contrast to nodes that do not provide their own access control, such as a printer, which provide a resource yet exert no control on its use, and assume users will always access it safely.

2.1 Model Overview

We first construct a model of a decentralized, distributed system and mechanisms used for implementing access control.

2.1.1 Definitions

Credentials

- A *credential* is a document used in deciding access control that can be transmitted over a network.
- A *trusted source* is a service (defined later) designated as a generator of credentials for a particular class or classes of credentials.

- A credential is *valid* only if it originates at a trusted source for its class of credential. A credential is *invalid* only if it is not valid.
- A credential is *verifiable* only if it is valid and there exists a function or method for any service to determine that validity. A credential is *intrinsically verifiable* only if that function or method is locally computable.

Services and Networks

- A *service* s is a participant in the distributed system, represented by a 3-tuple (S, I, A) where:
  - S is a state machine governing the behavior of the service.
  - I is an *interface* composed of a set of *operations* $O = (F, P)$ where F is a function name, and P is an implementation-dependent set of types specifying the parameters over which the function named by F is defined. Each O is a possible state transition in the state machine S.
  - A is an *access control policy* that decides whether s will allow activation of an operation by another service. If C is the set of all possible credentials, A is itself a function $O \times C \to \{\text{true}, \text{false}\}$ that maps an operation and a provided credential to the access control decision.
- A *client* is itself a service, but specifically in the context of an interaction is used to distinguish the service making a request from the service answering the request.
- A *component* is a service that is not a human operator, but rather a program or programs operating on a computer.
- A *network* N is the graph of all services s and bi-directional communication channels between services. When there exists a path of channels between two services s, t we say that s, t are *connected*. We assume a connected channel provides reliable message delivery, but with no guarantees on timeliness.
- An *adversary* is a service whose goal is to successfully gain access to an operation to which it should not have access. In service-oriented systems, adversaries are other services which attempt to subvert the system for their own gain.

The problems of message integrity and privacy have been well-studied both in theory and in practice, and therefore we will assume adversaries cannot undetectably modify messages in transit or eavesdrop on secured channels. Decentralized systems of services use these facilities for providing these guarantees, and we will make use of them in our implementations. We focus on the role of services as adversaries when working within the system to obtain this improper access. We also assume adversaries cannot easily create new identities for themselves, so as to launch a Sybil attack [48].

Interactions

- An *access* to a service e is the submission to e of a tuple $a = (o, c)$ where o is the instantiation of an operation in O, with a particular named function and parameter values, and c is an access control credential.
- An *access control system* for a distributed system is the set of all access control policies A of the services in the network, which together implement a desired security policy such that:
  - For each A, A evaluates to true only if the requesting service is authorized according to the desired policy for the requested operation, and to false otherwise.
  - For any credential c, either c is verifiable or c is not valid.
- *Cooperation* is the process by which a service acts towards completion of a request made by another service.
- *Coordination* is the process by which two or more services use communication to ensure a series of events occur that require cooperation.

Capabilities

- A *capability* is a self-validating credential that authorizes access to a resource.
- A *contractually-limited capability* is a tuple $C = (o, \mathcal{C})$ where o is an operation on a particular service s, and $\mathcal{C}$ is an operation-dependent *contract* used in the computation of A, the access control policy function. A capability is

a credential. A contract is a set of constraints on the use of a capability. C *satisfies* an access a if the operation in a is the same as the operation o authorized by C, and the contract $\mathcal{C}$ is satisfied.

- In a *capability system*:
  - The only trusted source for a capability is the service implementing the operation authorized by that capability.
  - All capabilities are intrinsically verifiable.

  An access control system which makes use of capabilities implements its decision function A as follows:

$$A = (a, C) \mapsto \begin{cases} \text{true} & \text{iff } C \text{ satisfies } a \\ \text{false} & \text{otherwise} \end{cases}$$

  Each capability C is created by the providing service, which is responsible for ensuring C cannot be forged or altered. A is therefore the verification that C is valid, should an adversary attempt to express a request for which it does not have a capability.

  In a capability approach, authority is delegated by sharing the capability with another service.

- A *session* is the series of accesses made using a particular capability issued by the providing service. The session ends when that capability is never used again. This could be due to its expiration, or deletion of all copies.

  A session may involve requests from multiple clients, in the event capabilities are delegated; in this way a session is defined by the interaction between a service and a particular capability, rather than the interaction between a service and a particular client.

2.2 Associative Broadcast

Associative broadcast is a model of coordination used in the CoorSet system in chapter 4. Associative broadcast's addressing is similar to content-addressing of

associative memory. A target set specification travels with each message that is broadcast onto the network. For each message, the target set is the set of recipients whose local states match that of the target specification. Therefore, the sender does not need to know the identities of the targets in order to broadcast a message; the cardinality of the target set is completely transparent to the sender, even when the set is empty. Associative broadcast does not assume messages arrive in the same order at each service.

We give here the definitions relevant to associative broadcast. The full treatment of its use as a coordination model is given in chapter 4.

Descriptive Names. Associative communication is based on descriptive names. *Descriptive names* carry information about the state of objects to which they are bound.

Profiles. Descriptive names are implemented as profiles. A profile is a table of attributes and attribute type/value pairs, which describe the state of a process or object.

Selectors. The target set of receivers for each message is determined by a selector, which is an expression in propositional logic over the attribute domain. The conditional expression of a selector is a template with slots for attribute name and value pairs specified in the profiles of other object instances. A selector is said to match a profile whenever the conditional expression of the selector evaluates to true when the attribute values from the profile are inserted into or compared with the slots in the selector expression.

Domains and Attributes. Interacting processes have *a priori* agreement on the set of attributes, which will appear in the profiles of components and the selectors of messages and the meanings of that set of attributes. The *domain* is the set of all possible attributes.

Messages. A message consists of a selector and content. The content of a message is implementation dependent. Coordination and composition require that the message specify an action to be executed by the receiving components.

Message Receipt. The selector of each broadcast message is locally matched with the profile of each object in the broadcast domain. Messages are received

only by those components whose profiles cause the selector of the message to evaluate to true. Matching of a selector to a component profile is atomic with respect to a profile's modification. It is therefore possible for a sender's target set to change during transmission if a profile is changed, but the matching criteria will not change for a component once matching begins. Matching is done locally and asynchronously upon arrival of the message.

2.3 Related Work

Decentralized, distributed systems of services arise from peer-to-peer systems, grid computing systems, and web services. Examples of peer-to-peer systems are Chord [116], CAN [105], Pastry [107], Tapestry [132], and BitTorrent [38]. Grid computing systems [53, 54] have begun to support multiple administrative domains and services architectures [59]. Web services [129] originally required the service to implement its own ad-hoc access control, but standards like WS-Security [92] have begun to provide standard mechanisms.

Capabilities first appeared as a mechanism for privileged invocations in multiprogrammed computations [43]. They gained widespread exposure as the authorization system for files in the HYDRA operating system [39]. They have since appeared in other operating systems [80, 81, 115] both as the access control mechanism for privileged system calls and file access, and for remote procedure calls between hosts. A taxonomy of capabilities [71] has been proposed in response to the inability of capabilities to enforce the *-property [26]. We extend the notion of capabilities here to support their use with a limiting contract, and for use in a networked system. We will examine these applications in depth in chapter 5.

Associative broadcast was originally formulated by Bayerdorffer in [11, 12], extended by Browne, Kane, and Tian in [30], and again by Kane and Browne in [74] for use in the CoorSet system. These extensions will be presented in chapter 4.

2.4 Conclusions

This chapter has presented the model of distributed systems we wish to examine. In the next chapter, we examine the fundamental issues of access control in decentralized systems, and the new challenges that arise with the lack of central control.

This results in a greater role of access control beyond simply assuring selected users have access to selected resources. Access control plays an important role in the central function of these systems. We will then return to this coordination model in chapter 5, taking the lessons learned in the taxonomy to devise an access control implementation for CoorSet.

Chapter 3

A Taxonomy of Access Control

3.1 Introduction

With the model of decentralized systems now established, we can proceed to examine the issues and design choices for implementing access control in these systems. An access control policy is typically a declarative specification of allowed accesses to resources[1]. An access control scheme, as defined by [121], is a state transition system in which access control decisions are specified as changes of state which conform to the stated access control policy in an appropriate representation such as an access control matrix. An access control implementation is a realization of a scheme on a networked system of services. Access control schemes are typically formulated under the assumption that decisions are made using complete and consistent state. This assumption may not be valid for a distributed system and is certainly not valid for distributed systems without central control, such as peer to peer networks. Therefore, implementations of access control for distributed systems may not faithfully conform to the access control scheme. There is a need to understand the trade-offs among fidelity of an implementation to a scheme, performance and scalability of the implementation and possible vulnerability of the implementation to attacks. This goal of this paper is to contribute to understanding these trade-offs.

Access control systems are commonly classified by the abstraction used to design the policy, such as user-based, role-based, or mandatory access control, by

[1]Please refer to section 3.3 for more detailed definitions of access control policies, schemes, models, and implementations.

an implementation mechanism such as access control lists or capabilities, or both. These classifications do not differentiate between centralized and distributed implementations and do not offer insight into the relative robustness of different implementations in the context of distributed systems. They do not define the space of possible distributed implementations. This paper presents an analysis of the properties of implementations of access control, particularly for distributed systems. The classification and analysis is based on identification of properties of access control system implementations and constructing a lattice taxonomy where each axis represents a property and the points on the axis are determined by possible values of the property. The systems analyzed are those given as examples in Section 3.5: Akenti [119], dRBAC [55], CRISIS [13], Kraft-Schäfer [78], BitTorrent [38] and WPA [127]. The current taxonomy has six axes representing: partitioning of control over sharing of access control credentials, distribution of the state relevant to access control decisions, fidelity of policy enforcement, the identity resolution mechanism, local versus centralized access control decisions and static or adaptive trust management. Each of these properties impacts the faithfulness with which an implementation implements a scheme and thus indirectly the vulnerability of the implementation to certain attacks. The current set of axes is not totally orthogonal but is still useful for classification, analysis of potential vulnerability to some forms of attacks, and suggestions for implementations with interesting properties which have not yet been analyzed or implemented.

The remainder of this chapter is organized as follows. We discuss other classification efforts and related work in section 3.2. We present definitions for access control in section 3.3. We then present the six classifying axes, each representing a particular property of access control implementations, and a number of discrete points on each that represent design or algorithm choices in section 3.4. We then classify the systems listed above in section 3.5. We discuss these results and note some unoccupied space in section 3.6, and conclude in section 3.7.

3.2 Related Work

We divide related research into two categories: surveys and descriptions of access control policies and models, and comparison of expressiveness among schemes and models.

3.2.1 Surveys and Descriptions

di Vimercati, Paraboschi, and Samarati [44] survey current systems in the single-host case and their access control implementations. They list desirable goals for access control systems, and then discuss the implementation of access control in Linux, Windows, Database Management Systems, TCPD, the Apache web server, and Java 2.

Sandhu and Samarati [112] discuss the Access Control Matrix, and implementations of both access control lists (ACLs) and capabilities. This discussion focuses mostly on the single-system case. They describe Discretionary Access Control (DAC) and Mandatory Access Control (MAC), both official standards from the U. S. Department of Defense, and introduce role-based policies which later led to role-based access control models [111]. Sandhu revisits these topics in [110], further developing the role-based access control model and introducing the Task Based Access Control model.

These papers provide useful descriptions of the space of policies, models and schemes and mechanisms for implementation but assume complete faithfulness of implementations with policies and/or schemes and do not enable comparison of implementations with respect to performance, scalability or security properties.

3.2.2 Comparison of Expressiveness of Schemes and Models

Tripunitara and Li [121] give a theory for comparing the expressiveness of access control models in terms of *simulatability*. It is their definition of an access control scheme that we have used in this paper. The existence or non-existence of a simulatability relation between schemes shows the relationship between the relative expressiveness of schemes.

Bertino, et. al. [18] formulate a logical framework for comparing the expressiveness of access control models for data based management systems where each instance of the framework corresponds to a program in a logic programming language. They give a classification of models in terms of structural equivalence and in terms of reachability for a given state of a scheme and consistency among reachable states.

Comparisons of expressiveness among policies/models does not address differentiation among or comparison of possible distributed implementations of the

policies and schemes which is the subject of this paper.

3.3 Definitions

3.3.1 Access Control Policies, Schemes, and Implementations

The development of access control goes through three stages: formulation of a *policy*, representation of that policy as a *scheme*, and finally realization of that model in an *implementation*.

- An *access control policy* is a definition of how a system should provide or deny access which can range from an abstract statement like, "only users on this list should have access," or "only users who have given me service in the past should have access," to *policy languages* with executable (operational) semantics [22, 24, 36, 83].

- An *access control scheme* [121] is a state transition system $\langle \Gamma, Q, \vdash, \Psi \rangle$ that embodies the access control policy. Γ is a set of states, Q is a set of queries that include privileged requests considered by the system, $\vdash: \Gamma \times Q \rightarrow \{true, false\}$ is the entailment relation that determines whether a given query is true or not in a given state, and Ψ is the set of state-transition rules[2]. $\gamma \vdash q$ means that the query q is true in state γ, and $\gamma \nvdash q$ means that it is not. In particular, when q is a privileged request, $\gamma \vdash q$ means the request is allowed in state γ.

- An *access control implementation* is the realization of such a scheme on actual hardware on a service i. In the distributed systems we consider here, each state $\gamma \in \Gamma$ is the global state of the entire system, made up of the *local states* $(s_1, s_2, \ldots, s_n)$ for each service in the system, where each $s_i \in S_i$, the set of all possible states for a particular service i. Global state may be replicated across multiple local states, and some of these replicas may become outdated. A local state is always dependent on the global state, but for brevity we will assume local states are in the context of the currently-mentioned global state. When more than one global state is mentioned, we will annotate the local states with their corresponding global state: for example, $s_{i(\gamma_1)}$ is the local

[2] [121] also defines an access control model as a collection of schemes. This definition of an access control model is not relevant for the purposes of this chapter.

state of service i in global state γ_1, and $s_{i(\gamma_2)}$ is the local state of service i in global state γ_2.

The implementation computes a function $\vDash$: $S_i \times R \rightarrow \{true, false\}$, where S_i is the set of local states of the service, and R is the set of specific requests being considered. We specifically note that $R \subseteq Q$: a specific request for service is always a type of query, but it is possible the scheme can answer more general kinds of queries. Generally, a service will only decide requests that it is asked to perform, although in the case of authorization servers it is possible the decider is not the servicer.

The goal of an implementation is to follow the scheme as closely as possible. Typically, the queries in a scheme are formulated so that they can be implemented perfectly, or nearly perfectly, if the relevant system state is complete and consistent. For some systems with distributed control, the nature of the system prevents perfect correspondence, which gives rise to this distinction between a scheme and its implementation.

3.3.2 Implementations versus Schemes

We now address the case where a scheme and an implementation can differ. A scheme can consider the state of an entire networked system, whereas an implementation may have to make a decision on the basis of partial and/or inconsistent state. We say that a request r is *improperly serviced* if for a given service i, and its local state s_i in a particular global state γ, $\gamma \nvdash r$ but $s_i \vDash r$, and so the service grants the request when it should be denied.

Similarly, we say that a request is *improperly denied* when $\gamma \vdash r$ but $s_i \nvDash r$, and so the service denies the request when it should be granted. We call either of these situations *improper*. An improper request is *detectable* if the implementation can transition to a state such that the impropriety of the request becomes known. Such a request is *always detectable* if the implementation is guaranteed to transition to such a state.

3.3.3 Access Control Credential

An access control *credential* is a bitstring used in access control decisions. It is protected from undetectable modifications through a mechanism such as cryptography

or kernel-imposed restrictions.

3.3.4 Lattice Taxonomy

We define a lattice to be a set of points on a set of axes. Each axis represents an enumerated type. An ordering metric can be defined for each axis. A lattice taxonomy classifies systems by assigning each system a value for each of the axes of the lattice. An ordered lattice taxonomy has one or more metrics which order points on the lattice globally.

3.4 Classification Axes

We now present the axes we have identified from access control implementations. We identified these axes by formulating implementation of access control as a workflow, identifying the steps in the workflow and then analyzing a number of access control implementations for networked systems, and extracting properties of the implementations of each step for these systems. The workflow formulation we used has the steps: (1) authentication of identity, (2) acquisition of system state information, (3) the access control decision process and (4) enforcement of the decision.

3.4.1 Control over Sharing of Access Credentials

The holder of an access control credential may be able to share access to a service by delegating that credential to another client while the service which granted the access credential may be able to revoke the access authorized by the credential. Consider two clients i, j and a service k (with local states s_i, s_j, s_k respectively), and requests r_i that states "i requests service from k", and r_j that states "j requests service from k". Suppose there is a reachable global state γ such that $\gamma \vdash r_i$, but $\gamma \nvdash r_j$, and also $s_k \vDash r_i$ but $s_k \nvDash r_j$. Let C be the credential(s) used by i to make the request r_i, and let the statement "i delegates C to j" indicate the invocation of a state transition ψ such that s_j now contains C. Let $\gamma \mapsto_\psi \gamma_1$ denote ψ causes a transition from global state γ to γ_1.

An access control credential is:

1. *Delegatable*, if ψ also causes a change of state $\gamma \mapsto_\psi \gamma_1$ such that $\gamma_1 \vdash r_j$, and $s_{k(\gamma_1)} \vDash r_j$, meaning that such a delegation then permits r_j,

2. *Revocable*, if from γ where $\gamma \vdash r_j$, there exists a reachable state γ_2 such that $\gamma_2 \nvdash r_j \Rightarrow s_{k(\gamma_2)} \nvDash r_j$, meaning the implementation can support the revocation of an authorization, and

3. *Fine-grained revocable*, if from γ where i and j are both authorized, there exists a reachable state γ_2 such that $\gamma_2 \nvdash r_j$ but still $\gamma_2 \vdash r_i$, the implementation can ensure that $s_{k(\gamma_2)} \nvDash r_j$ while still $s_{k(\gamma_2)} \vDash r_i$. The absence of this third property means that i's authority must be revoked in order to revoke j's, assuming such authority can be revoked at all. This can only apply where property 2 applies.

We identify the following points on this axis, ordered by increasing amount of control given to the service:

1. **Client control.** The client can delegate credentials at will once the service provides it, and that credential cannot be invalidated. Therefore, delegation causes a state change that grants access to the delegate j. Credentials are neither revocable nor fine-grained revocable.

2. **Shared control without recorded delegations.** Credentials are delegatable, but the service can transition to a state where the access is revoked. Clients still can freely delegate their authorizations. The service is able to revoke that authorization and all delegations made of it. Access credentials are revocable but not fine-grained revocable.

3. **Shared control with recorded delegations.** The state change ψ caused by the delegation also causes some state to be added that tracks delegation. This is typically achieved by requiring an addition to the credential to delegate it. This state may not be recorded on the service itself, but eventually becomes available to the service. This now gives the service additional information, which may allow it to revoke authorizations in a more fine-grained manner, possibly by invalidating all credentials and re-issuing credentials to only the desired recipients.

4. **Service control.** Credentials are not delegatable at all. This allows them to be completely revocable and fine-grained revocable, as there is a 1:1 correspondence between credentials and clients authorized directly by the service.

The points on this axis define both a metric for control of access and a trade-off between complexity of processing of queries/requests and ability to support complex collaborations among clients. Shared control with recorded delegation may require complex processing to validate a delegated credential.

We are unaware of any implementations at the user control point, but it does represent the absolute minimum of control given to the service. We will see examples of services at the other three points in section 3.5. We specifically ignore the possibility of sharing an identity credential for the purposes of impersonating another client as a means of delegation; we instead will address this issue in section 3.4.4.

3.4.2 State Distribution

Any access control implementation must maintain information relevant to access control decisions. Credentials transmitted across a network are just one type of such state. For example, this information can also be access lists, capability lists, historical data, or any other state needed to compute the access control decision. This axis represents how the global state γ is distributed across the local states $s_1, \ldots, s_n$. We specifically examine the case where for a particular request r "i requests service from k", how much state information is kept in s_i versus the amount kept in s_k for only that particular request. We are therefore not considering the aggregate state of k for all requests it considers. We also do not count any service-wide state it maintains, such as its own identity certificate, or list of operations provided.

We present the following points on this axis, listed in increasing amount of state per authorization stored by the client i. When state is stored across multiple physical sites on the serving side, such as when a service queries a trusted server for the access control decision, we group all of this state under the management of "the service." Let $\mathcal{S}$ be the space requirement of the service, and $\mathcal{C}$ that of the client.

1. **Centralized.** $\mathcal{S}$ is unbounded, and $\mathcal{C} = 0$. The service maintains all state information, and the client stores zero state – not even a username or password.

2. **Service managed.** $\mathcal{S}$ is unbounded, and $\mathcal{C} = O(1)$. The service maintains that part of the access control-related state, which may grow to arbitrary size, while the client stores constant-sized state, such as a username/password pair.

3. **Equal sharing.** $|\mathcal{S}| \approx |\mathcal{C}|$. The service and the client each maintain arbitrary per-authorization state. There may be differences in size or content.

4. **Client managed.** $\mathcal{S} = O(1)$, and $\mathcal{C}$ is unbounded. The service stores constant state for each authorization, but the client's storage requirement may grow arbitrarily large.

5. **Decentralized.** $\mathcal{S} = 0$, and $\mathcal{C}$ is unbounded. The service stores zero state, and so the client stores all access-control state.

This axis is related to vulnerabilities to different attacks and to the cost of maintaining consistency between local and global state. State kept on clients minimizes the impact of a successful theft of data but increases the probability that an attack will be successful since clients may be less well protected than services. State kept on clients also requires that more state data be communicated for each access. If a service maintains access control state data on its clients then an attack may result in widespread damage.

The only centralized system of which we are aware is one that is anonymous, such as anonymous FTP, where not even a username and password are stored by the client to get access. We will consider systems at the other four points in section 3.5.

3.4.3 Fidelity of Enforcement

A distributed implementation of an access control scheme may not precisely match the behavior of the access control scheme it implements, since it may be prohibitively expensive to gather all of the necessary state to make perfect decisions. We quantify deviation from complete fidelity of an implementation to its scheme by introducing the enforcement axis, which measures how well an implementation follows the decisions of its scheme. We present the following points in decreasing amount of fidelity of the implementation to the scheme, as we expect the earlier points will be more familiar to the reader.

Recall that $\gamma \in \Gamma$ is the global state of the system, and s_i is the local state of a particular service i in that state. Further, recall that $\vdash$ is the query result according to the scheme, which has perfect knowledge of global state, and $\models$ is the query result according to the implementation, which has only the local state as maintained by

the implementation to consider. Let r be a request under consideration. These properties are taken for all states γ and the corresponding s_i in that global state.

1. **Total fidelity.** In this case, $s_i \vDash r \Leftrightarrow \gamma \vdash r$. No improper requests can occur.

2. **Total prevention.** A slightly weaker case, $\gamma \nvdash r \Rightarrow s_i \nvDash r$. A request may still be denied when it should be granted, but a request will never be granted when it should be denied.

3. **Total detection.** In this case, it is possible that $\gamma \vdash r$ but $s_i \nvDash r$, or $\gamma \nvdash r$ but $s_i \vDash r$, meaning that the service makes an incorrect decision due to its incomplete state data, so we have only partial fidelity. However, improper requests are *always detectable*, so a service eventually discovers when a deviation from the scheme has occurred.

4. **Partial detection.** There is still only partial fidelity, but now improper requests are not always detectable: a service is not guaranteed to later detect an incorrect decision. Improper requests still are detectable, so there is a chance the violation will be discovered.

5. **No detection.** In this situation, improper accesses are not even detectable; a service makes a decision and cannot later tell whether or not it was correct.

6. **No fidelity.** In this situation, $(\forall r) s_i \models r$, regardless of $\vdash$. This is the absence of an access control mechanism.

This axis is a direct metric for strength of compliance to the scheme being implemented. The complexity and cost of attainment of each level of fidelity in a distributed implementation depends upon the queries permitted in the scheme and the cost and complexity of maintaining complete and consistent local state information. In the Kraft-Schäfer [78] mobile ad-hoc routing system (see section 3.5.4) queries concern forwarding messages in an ad-hoc network. The dynamic network structure causes transient incompleteness of state information inducing a window of possible loss of fidelity. BitTorrent [38] (see section 3.5.5) has a similar window of fidelity loss between when a client decides not to cooperate, and when that lack of cooperation is detected.

3.4.4 Identity Resolution

Although identity does not have to be used in an access control decision, it is done with such frequency that the mechanisms by which identity is established should be a part of any characterization of access control implementations. At some point in the operation of such a system, a service must make a connection between a client outside of itself and a local representation. In simple systems, it may be the connection between a client and a particular set of permissions. In role-based systems [55, 111], it may be the connection between a client and a role it is attempting to assert. The connection that spans the gap between outside the service and inside the service is called the client's *identity* and is usually represented by an *identity credential*. While a scheme assumes that all identity credentials which are presented are valid, this cannot be assumed in an implementation. The confidence in this connection measures the likelihood of convincing an outside party, such as an impartial judge, of the strength of the binding between that representation and the outside party. We present the following points in increasing order of confidence given to this binding.

1. **No identity mechanism.** This is the lack of an identification, where all clients are classified exactly the same.

2. **Shared secret.** In this mechanism, clients are given a shared secret to distinguish them as members of a selected class, distinguishing them from clients who do not possess the secret. There is no differentiation amongst individuals in the group; merely a classification into authorized and not authorized.

3. **Known name and shared secret.** This adds to the shared secret by differentiating amongst individual clients. This can allow a service to introduce different levels of authorization.

4. **Known name and certificate.** A certificate is defined as a document attesting to the truth of some fact. An identity certificate specifically attests to the identity of the bearer – the very connection we wish to make when establishing identity. This certificate adds to the confidence by adding the testimony of a witness or authority.

5. **Unforgeable identity.** There is a fool-proof guarantee of identity, or identity

is established through a mechanism whose probability of error can be safely neglected, such as biometric identification of human operators.

This axis is a metric for vulnerability to attacks since many attacks are based on an adversary successfully impersonating another client.

3.4.5 Decision Mode

In a distributed system, the ultimate authority of whether to provide a service is the service itself. Services may consult other services, or defer their decision to a chosen authority. For any request r "i requests service from k" under consideration, we define two subsets of the services' state machines: r_{IN} which is the set of local states that contribute input to the decision on r, and r_{OUT}, which is the set of state machines that have control over whether or not $s_k \vDash r$.

We present the following points in increasing order of how globally such decisions are made.

1. **Local.** The simplest case, where an access control decision is made based entirely on the local state of the service. $r_{IN} = r_{OUT} = \{s_k\}$.

2. **Advised.** In this case, the ultimate decision is still made by the service itself, but it reaches out to other services to collect information, to get a better view of the global system state and thus make a more well-informed decision. $r_{OUT} = \{s_k\}$, but now $|r_{IN}| > 1$. Generally, $s_k \in r_{IN}$ as well, but this is not required.

3. **Consensus.** In this case, not only are the opinions of other services involved, but some group collaborates following an agreed-upon protocol and reaches a common decision, with none having ultimate authority. In this case, both $|r_{IN}| > 1$ and $|r_{OUT}| > 1$. s_k is often, but not necessarily, in each set.

4. **Centralized.** A special case of consensus, and the most global decision, where the "group" is a single authority that is not the service itself. Here, the central authority A has the best possible knowledge of the global state, and so can make the best decisions. There is no restriction in the size or membership of r_{IN}, although typically it consists of just s_A. However, $r_{OUT} = \{s_A\}$, and also, $A \neq k$.

Schemes are defined on the assumption that decisions are made on the basis of complete and consistent global state. Any distributed implementation is a trade-off between cost of state maintenance and fidelity. The points on this axis represent trade-offs between fidelity, scalability and performance and cost of maintenance of complete and consistent state.

3.4.6 Static or Adaptive Trust Management

"Trust" is shorthand for a standard of expected behavior. The choices of whether to incorporate trust in an access control decision and whether or not trust is statically assigned or dynamically computed on the basis of runtime behavior is a property of a policy and thus the scheme representing the policy. We include static or adaptive trust management as a property of an implementation since dynamic computation of trust is a source of loss of fidelity of an implementation with respect to a scheme. For most systems with centralized access control, trust relationships do not change without the outside intervention of an operator. With increasingly distributed control, a service may need to revisit its trust relationships during execution to respond to changing conditions, such as evidence that a service has altered its behavior. We specifically exclude the case where an administrator or outside influence causes changes to trust relationships, as we instead view this as the state machine itself being altered; here we address only when there the service can transition between states that reflect a change in certain trust relationships as a result of run-time events. There are two cases:

1. **Static** (or manually changed). Trust relationships never change as a result of run-time relationships, and remain static throughout the operation of the system, excluding changes effected externally. Therefore, if $s_i \vDash r$ in *any* global state γ, then $s_i \vDash r$ in *all* γ; and similarly for $s_i \nvDash r$.

2. **Adaptive.** $(\exists r, \gamma_1, \gamma_2) \left[s_{i(\gamma_1)} \vDash r\right] \wedge \left[s_{i(\gamma_2)} \nvDash r\right]$. The service can transition to states that carry an altered trust relationship, as a result of run-time events, such as internal state changes, the behavior of clients, or other aspects of interactions with other services.

Implementations utilizing adaptive trust relationships can constructively respond to certain forms of attack or access abuse.

3.5 Example Classifications

This sections gives a brief characterization of each system in terms of what kinds of requests its scheme and implementation support, followed by its classification under each axis. For brevity, we do not give a complete description of the access control scheme, but instead give a brief characterization of each system in terms of what kinds of requests its scheme and implementation support, followed by its classification under each axis.

3.5.1 Akenti

Akenti [119] is a system for access control in grid environments. Each client has an identity certificate, and *stakeholders* who control resources express their rules for access in certificates they sign and provide. In this system, servers providing resources look to a trusted Akenti policy server to make the decision, rather than making its decision locally. Requests in this system concern only accessing grid resources, subject to those restrictions. The properties of Akenti are as follows.

- Control over Sharing: **Service control.** Clients cannot delegate any authority to other clients. Authority must come directly from the stakeholder's access control description.

- State Distribution: **Service managed.** Clients each store an X.509 certificate, and that is all. The resources and policy servers, which we consider both part of the service side, store all the stakeholder-provided access control restrictions.

- Fidelity of Enforcement: **Total prevention.** Requests are always evaluated whether or not they follow the provided stakeholder certificate. A request will only be granted if the resource server can contact a policy server and get a positive answer. If a policy server cannot be contacted, the request will be denied, which may be at odds with the intended behavior. But, a request that should be denied will always be denied.

- Identity Resolution: **Known name and certificate.** An X.509 certificate, that must be signed by a recognized Certifying Authority, is used for identification. Akenti uses commercial vendors such as Netscape, Entrust, or Verisign

for signed identity certificates. Akenti uses TLS [45] as the authenticating transport.

- Decision Mode: **Centralized.** Decisions are always deferred to Akenti policy servers. The policy server may be replicated, and so there may be several, but it is in effect one single decision-maker.

- Trust Management: **Static.** All access control restrictions are statically assigned by stakeholders. This will never be changed by the system itself.

3.5.2 CRISIS

CRISIS [13] is the security architecture of the WebOS [124] distributed operating system, which uses many of the ideas of the shared-memory multiprocessor operating system TAOS [4]. CRISIS introduces an architecture of *principals* to represent clients, *objects* to represent services, and *resource monitors* to represent access control decision-makers. Clients have identity certificates, and can construct *transfer certificates* to give to other clients to delegate portions of their access. Transfer certificates contain predicates that limit the access delegated. Access is granted if a client presents an identity that is explicitly authorized for a resource (by appearing on the resource's access list), or presents a chain of transfer certificates proving delegation from such an authorized client and satisfaction of all limiting predicates on all transfer certificates. Requests concern access to arbitrary services available in the network. The states of the system incorporate transfer certificates which record delegations that have been made. The properties of CRISIS are as follows.

- Control over Sharing: **Shared control with recorded delegations.** A client makes transfer certificates to transfer a subset of its privileges to another. This results in a chain of certificates refining a set of capabilities to the point where the principal attempting to use them is found, and are used as proof of access rights. When presented for invocation, this information becomes available to the service, which can be used for later changes in access control policy or system auditing.

- State Distribution: **Client managed.** A service must store an entry in an access control list for each authorization, with a list of access rights. A client

wishing to make use of a resource must store a certificate chain, which can grow to arbitrary size depending on the number of delegations made, and present it to make an invocation.

- Fidelity of Enforcement: **Total fidelity.** As the access control policy is encoded directly into the reference monitor, and the possible queries are all permitted or not directly based on that policy, the reference monitor can perfectly implement the scheme.

- Identity Resolution: **Known name and certificate.** CRISIS uses two kinds of certificates: identity certificates and transfer certificates. The identity certificates are signed by both a trusted certifying authority and an on-line authority.

- Decision Mode: **Local.** The resource monitor needs only verify the chain of signatures, and then verify the root of the chain is an explicitly-authorized client on its local access control list.

- Trust Management: **Static.** Changes to access control restrictions are only ever made by explicit alteration of access control lists by an operator.

3.5.3 dRBAC

Distributed Role-based Access Control (dRBAC) [55] is a distributed access control system for managing systems which have components in multiple administrative domains. It incorporates the RBAC model in [111], and includes delegation chains similar to those in CRISIS. They abstract both clients and resources in the system as *entities*, each of whom has its own local namespace. It reconciles these two features by introducing delegation semantics that delegate roles to other roles or entities. Only roles can be delegated, not individual rights. These delegations can be extended with *valued attributes* which further limit the access level. The right to delegate is itself a right that must be delegated separately; being delegated a right does not allow one to delegate it further unless the right to delegate has been delegated as well.

In dRBAC, an entity is the only certifying authority for entries in its namespace, which corresponds to resources being the only ultimate authority for operations

they export, and so each chain must end with a delegation by the entity itself. These delegation chains are then verified by *proof monitors*, which is part of the dRBAC infrastructure, present at each entity. In this scheme, requests concern access to specific resources, and the pertinent states of the system concern what delegations have been made between clients. The properties of dRBAC are as follows.

- Control over Sharing: **Shared control with recorded delegations.** dRBAC uses a certificate chain where each link in the chain is a statement of the form `[Subject → Object] Issuer`, where an Issuer asserts that the Subject has the role of Object. Subject can be a role or an entity, but Object is always a role. This allows assigning roles to entities, as well as roles to other roles.
- State Distribution: **Decentralized.** These certificate chains are maintained by the clients and presented when service is requested. The proof monitor needs only to be able to verify the chain of signatures, which it can do with just the submitted chain, and it needs no per-authorization state to do so. Once the delegation is extended, it can then be forgotten by the service.
- Fidelity of Enforcement: **Total fidelity.** As the requests are straightforward enough that they can be programmed definitively into a service, the service can follow the policy with total fidelity.
- Identity Resolution: **Known name and certificate.** Each certificate chain starts with an identity certificate, and then has zero or more delegation certificates. dRBAC assumes that when a delegation certificate is signed, the issuer is also attesting to the identity of the entity or role to which authority is delegated, eliminating the need for a separate certifying authority.
- Decision Mode: **Local.** All certificates can be validated without any external sources, and the final entry must be issued by the servicing entity, which can naturally verify its own signature.
- Trust Management: **Static.**

3.5.4 Kraft-Schäfer Mobile Ad-Hoc Networks

Kraft and Schäfer [78] give a system for the exclusive purpose of granting or denying relay access in a mobile ad-hoc network. To adhere to their nomenclature, in

this section we will specifically refer to "nodes" rather than services. The scheme adopts a trust metric to decide whether or not another node has been sufficiently cooperative and trustworthy to forward its packets; in a mobile ad-hoc network, the primary improper behavior is greedily using the network, and not forwarding the packets of others. Each node generates its own identity certificate, and can be vouched for by other nodes in a "web of trust." Both opinions of other nodes and observed behavior affects a node's local rating. The requests supported by this system are to request a packet be forwarded onward in the network, and the states concern historical data, and whether or not a particular node is trying to be rogue. The properties of Kraft-Schäfer are as follows.

- Control over Sharing: **Shared control with recorded delegations.** If one node decides it no longer can trust another node, and so no longer wants to pass traffic on that node's behalf, it can immediately cease. For the purpose of introducing new nodes to the network, established nodes can provide "warrants" to new nodes, to delegate part of their good rating to the new node.

- State Distribution: **Equal sharing.** All nodes in the network retain the same amount of information about each other, in tables of trust values.

- Fidelity of Enforcement: **Partial detection.** This system uses a heuristic to decide beyond what point a node is considered sufficiently uncooperative to warrant future denial of access. Although in the global state we can tell when a node decides to become rogue and not cooperate, it is not guaranteed that an individual node will detect this ever, depending on how uncooperative the rogue node becomes. At least one improper request can certainly be accomplished by the rogue node, as behavioral information is not available until after the improper request has taken place.

- Identity Resolution: **Known name and certificate**. In this case, to avoid Sybil [48] attacks but not require a globally-recognized certificate authority, certificates must be signed by another node in the network. The trust given such a certificate depends on its signer.

- Decision Mode: **Advised.** An individual node decides whether or not to forward traffic at the behest of another node. It may use input from other

nodes under a form of opinion sharing [67], but the decision to forward or not is entirely its own.

- Trust Management: **Adaptive.** Trust relationships are constantly evaluated by the system itself based on previous cooperation.

3.5.5 BitTorrent

BitTorrent [38] is a file distribution system used to spread the bandwidth cost of disseminating large files across a large number of nodes. As above, we use the term *node* here to be consistent with this work. It uses access control to enforce collaborative behavior. A node grants access to those who are providing data, and denies access to those who are attempting to consume without providing. In this analysis we consider only the peer-to-peer file transfer protocol, and do not consider the "trackers" which are used for bootstrapping. The requests are for downloading "chunks" of files, and the states of the system are the recorded history made by each service in its recent interactions with a particular client. The properties of BitTorrent are as follows.

- Control over Sharing: **Service control.** A BitTorrent peer serving a file decides directly what access to give a client. That access cannot be then shared with other clients.

- State Distribution: **Centralized.** A service maintains historical data on clients, but indexes them by network address as opposed to any independent identity. A downloading client has to find the service, but needs store no access control-related state.

- Fidelity of Enforcement: **Total detection.** A service can detect when a downloading client is not giving "tit-for-tat," and deny access after the fact. However, such a lack of cooperation will always be detected in time for the service to cease its own cooperation.

- Identity Resolution: **Unforgeable identity.** BitTorrent identifies a peer exclusively by its endpoint IP address and port number for the duration of its connection. It is unforgeable because as soon as a peer disconnects, its identity

and history is forgotten, in a way that makes it unforgeable by design.[3]

- Decision Mode: **Local.** The access control function is evaluated solely on information observed by the service.
- Trust Management: **Adaptive.** BitTorrent regularly re-evaluates the access control function to consider new observations on downloading peer behavior. It quickly chokes off uncooperative peers, and gradually increases cooperation with cooperative peers.

3.5.6 Wi-Fi Protected Access in IEEE 802.11 Wireless Networks

Wi-Fi Protected Access [127], or WPA, and its successor, WPA2, are access control mechanisms for 802.11a/b/g wireless networks. The requests handled by this scheme are to gain access to a wireless network for communication. It incorporates a Temporal Key Integrity Protocol (TKIP) which dynamically changes keys as the system is used.

The differences between WPA and WPA2 are not relevant for this discussion. However, WPA and WPA2 each operate in two primary modes: WPA-Enterprise (or WPA2-Enterprise), for use in a large-scale networks with an IEEE 802.1X-compliant authentication server, and WPA-Personal (or WPA2-Personal). Each mode has different properties under our classification, and so we will present each one separately.

Enterprise Mode

In the enterprise environment, a client possesses a set of credentials in the form of a digital certificate, username/password pair, smart card, or any other identity mechanism desired by the administrator. A client then authenticates to an authentication server, which is a single point for all wireless access points, which are the services. After this point, a shared session key is negotiated to allow access. The properties of WPA in this mode are as follows. The requests are the association/authentication protocol, and a request to forward messages. The relevant state is the authentica-

[3] It is theoretically possible for an intruder host on the same subnetwork or even the same host as a downloading client to simultaneously shut the peer out while taking its place, thus assuming its identity and its (presumably) favorable history. We are unaware of any such attack in existence and expect it is highly impractical to mount.

tion information programmed into the central server, and the state stored in the wireless access points to track what clients are authenticated at a particular time.

- Control over Sharing: **Service control.** Clients cannot delegate their authorization to other clients by any means.
- State Distribution: **Service managed.** A client must store an identification credential of some kind, but the authentication server can store any number of access restrictions, such as to which access points a client may associate, time-of-day usage restrictions, or validity periods.
- Fidelity of Enforcement: **Total prevention.** Access to the wireless network is only granted when the authentication server returns a positive response, and the authentication server is programmed with exactly the desired access control policy. If the authentication server is unavailable, requests may be denied when they should be granted.
- Identity Resolution: **At least known username and shared secret.** Part of the WPA-Enterprise standard allows choosing the identification mechanism. Such a mechanism is at least a username and password pair, but can be a username and digital certificate, smart card, or any other secure identity mechanism. We are unaware of any application of WPA-Enterprise that uses less confident measures.
- Decision Mode: **Centralized.** Each service (the wireless access point) defers its decision to a central authentication server, which makes global decisions on what clients to allow and what clients to deny.
- Trust Management: **Static.** Although it is possible for an authentication server to be designed to automatically update its trust relationships based on run-time events, we are unaware of any such implementations.

Personal Mode

For "SOHO" (small office/home) users who do not have such an elaborate authentication infrastructure, WPA can also operate in the "personal" mode. Also known as *pre-shared key* (PSK) mode, an individual access point and any authorized clients

are given a passphrase to access the network, much the same way as with WEP [66], although the implementation of the cryptography has been improved to avoid WEP's well-published vulnerabilities [10, 21, 28, 125]. In this case, there is no central authentication server, and wireless access points do not coordinate with one another (although they can share the same passphrase). The properties of WPA in this mode are as follows.

- Control over Sharing: **Shared control with unrecorded delegations.** The passphrase is now a credential that can be delegated, and there is no tracking of its being passed from client to client. Access by all authorized clients can be revoked by changing the passphrase and delegating it again to authorized clients, but there is no method of more fine-grained revocation.
- State Distribution: **Equal sharing.** The service and the client each need only now store the passphrase, and any intermediary keys negotiated as part of the authentication protocol.
- Fidelity of Enforcement: **Total fidelity.** The query in this system is whether or not a client knows the passphrase. If so, access is granted. This query is accurately implemented by providing the passphrase to the access point as proof of knowledge.
- Identity Resolution: **Shared secret.** As previously mentioned, the only differentiation amongst clients are those that are authorized by virtue of knowing the passphrase, and those that are not.[4]
- Decision Mode: **Local.** The access point is manually programmed with the passphrase, and needs no outside communication to determine the access control decision.
- Trust Management: **Static.**

3.6 Evaluation

The insights which arose from development of the classification scheme and analysis of the several systems include:

[4]Most wireless access points implement additional access control measures, such as filtering by MAC address, but these are in addition to WPA, and not part of WPA itself.

1. Access control is conventionally thought of as implementing security and management of resources. But in fact, access control is being used to play several different roles in the systems we have characterized:

   - Security and control over resources: Akenti, WPA, CRISIS, dRBAC
   - Construction of virtual systems: Akenti
   - Enabling collaboration: CRISIS, dRBAC
   - Enforcement of collaboration: BitTorrent, Kraft-Schäfer

   These broader definitions of access control arise primarily from the intrinsic requirements of distributed control for collaboration to accomplish goals.

2. Positions in the taxonomy where we did not find implemented systems but where one might expect to find future distributed systems include those where Control over Sharing is at least Shared Control with Recorded Delegation, where Decision Mode is Advised or Consensus and where Trust Management is Adaptive.

3. In systems with more distributed control, such as Kraft-Schäfer in section 3.5.4 and BitTorrent in section 3.5.5, implementation decisions are made which appear to make the enforcement weaker, but these implementations are, in fact, the strongest implementations for the given scheme devised thus far. This dichotomy stems not from poor design choices or implementation, but that the access control scheme now must support queries that prove difficult to implement with fidelity in a way that gives acceptable performance. Enforcement with total fidelity or total prevention requires maintenance of up-to-date global state information that is infeasible for networked systems of arbitrary size. Just as the legal system contends with illegal acts in a very large population by employing what prevention is feasible, detection where it is not feasible, and then incentive to comply in the form of punishments, this analysis suggests these new frontiers in distributed systems with fully distributed control will require such hybrid approaches as we are seeing emerge.

3.7 Conclusions

We have presented a classification for access control implementations, and observed that new systems, with more distributed control, have access control schemes that are more difficult to implement efficiently with complete fidelity, and have led to the necessity of less "perfect" mechanisms that still function. We give this as a basis for comparing access control implementations, and as a beginning to exploring the deeper unifying concepts amongst access control implementations, and the interplay between a scheme and a corresponding implementation.

The next chapter introduces CoorSet, a framework for computation in decentralized, distributed systems that has no standard mechanism for access control. We then return to the lessons learned from the taxonomy in chapter 5, where a standard access control mechanism is used that occupies one of these unexplored spaces identified here.

Chapter 4

CoorSet: A Development Environment for Associatively Coordinated Components

4.1 Introduction

CoorSet is a framework for computation in decentralized, distributed systems that was previously reported in [74]. It is based on a model of communication and coordination called *associative broadcast*, originally formulated by Bayerdorffer [11, 12] and extended in [30]. In associative broadcast, a rich naming model is used that enables targeted multicast and therefore replication naturally where a broadcast mechanism is available. Later, when we convert this platform into a secure services platform in chapter 5, we will make use of this mechanism as the method of discovering other services.

The layout of this chapter is as follows. Associative broadcast as reported in [30] is introduced in sections 4.2 and 4.3. Its use as a coordination model is then explored in section 4.4. Extensions made to this model as reported in [74] are then given in 4.5, and formulation of algorithms is explored in section 4.6.

CoorSet is introduced and illustrated with two examples in section 4.7. A full treatment of the CoorSet interface description language is given in appendix A. Section 4.8 describes the supported platforms for automatically launching CoorSet programs. Section 4.9 describes the implementation mechanisms used to provide

the necessary reliable broadcast facility, and security measures used in the CSC launching mechanism described in section 4.8. Section 4.10 explores related research and section 4.11 concludes.

4.2 Broadcast-Based Coordination

There has been little research on coordination models and languages based on broadcast communication, despite how many networks intrinsically provide a broadcast capability, including such widely available systems as Ethernet, FDDI, and wireless. That broadcast enables consensus for asynchronous communication [122].

Except for Linda-based [57] coordination models and languages, there has been relatively little experimental or systems-oriented research on application of coordination models and languages. Experimental research is needed to establish a basis for application of coordination models and languages and to add credibility to the utility value of coordination models and languages. This chapter extends the coordination model based on associative broadcast from chapter 2 into a development environment for implementation of coordinating systems of processes, illustrates its applications and positions this research in the context of distributed and peer to peer systems research. The goal for the development environment is to facilitate experimental research on broadcast-based coordination systems. The principal artifacts of the development environment are: extensions to the previous associatively broadcast programming model to facilitate experiments and applications, an interface definition language for expressing associative interactions, a compiler for this interface definition language and an environment for instantiating and executing coordinating systems of processes.

Broadcast enables coordination based on every process in an interacting set locally maintaining common state necessary for collective decision procedures [46, 122]. Associative broadcast enables targeting of messages to processes in specific states and enables each process to select the properties of messages it will receive. Basing coordination on associative broadcast communication enables definition of multiple dynamic coordination subsets in a set of processes. Separation of message filtering from computation decreases the execution cost of coordination using broadcast and allows for specialization to specific algorithm requirements. Associative broadcast preserves anonymity similarly to tuple space-based coordi-

nation [57]. It enables transparent distribution and replication for fault-tolerance. In summary, associative broadcast enables fully distributed and fully symmetric coordination over dynamic sets of processes.

4.3 Associative Interactions

We now present a coordination model based on a special form of broadcast communication: associative broadcast [11, 12]. We describe the model in terms of parallel programming with regards to processes and objects in a computation space. Later we will relate this model to decentralized service-oriented systems. Broadcast enables coordination based on every process in an interacting set locally maintaining common state necessary for collective decision procedures. Associative broadcast enables targeting of messages to processes in specific states and enables each process to select the properties of messages it will receive. Separation of message filtering from computation decreases the execution cost of coordination using broadcast and allows for specialization to specific algorithm requirements.

Broadcast enables coordination over dynamic sets and preserves anonymity similarly to tuple space methods [57]. It enables transparent distribution and replication for fault-tolerance. Associative broadcast enables fully distributed and fully symmetric coordination.

There has been relatively little research on coordination models and languages based on broadcast communication [25, 27]. Yet many computer networks intrinsically provide a broadcast capability (Ethernet, FDDI, MAN, wireless networks based on cellular communication, satellite transmissions, etc.). Coordination of distributed components and processes using broadcast is thus becoming of greater interest.

4.3.1 Coordination and Composition under Associative Broadcast

Coordination is formulated by defining profiles, messages and protocols. The execution model is an eventually synchronous [49] receive/action (state machine) model. A component receives a message and takes some action (possibly null) in response to the message. Recall that the terminology of associative broadcast was introduced in section 2.2 on page 9. An algorithm or computation is specified by:

1. A set of attributes in which the profiles and selectors are specified.
2. A set of rules for binding of the state of a component to a profile.
3. A set of protocols by which interactions are executed including specification of message types, the selectors to accompany each message, the allowed sequences of messages and the responses to each instance of a message type which is received.
4. A state machine for each component which implements the coordination protocol.
5. An initial state for each component.

Each component executes in accordance with the rules and protocols. When we use associative broadcast as a discovery mechanism in chapter 5, we will relax this requirement to allow for Byzantine behavior. There is no coordinator or global control state. Coordination and thus control is fully distributed. If desired, coordination can also be fully symmetric with the deployment of the same state machine on each component. The set of components which are interacting and coordinating can be dynamic. A component can enter a network by setting an appropriate profile and possibly following some initialization protocol.

With associative broadcast, coordination is independent of process location within a broadcast domain. Coordination extends over dynamic sets of processes. Coordination decisions do not depend upon stable group properties.

4.3.2 Composition as Coordination and Vice Versa

Composition is defined in terms of binding invocations of a method or procedure by one component to a method or procedure in an implementing component. Although the mechanisms differ, composition may occur at compile time, at link/load time and at run time through a mechanism like remote procedure calls. Composition is intrinsic to associative interactions. Each message may result in binding of a message to a component or set of components when it is sent. Each binding of the message results in the execution of an action. Associative naming and binding can be used as to select an initial set of bindings among components prior to runtime during compilation. A compiler which uses associative interfaces to compose programs

implementing performance models from components has been reported [29]. The associative model of interaction integrates coordination and composition.

4.4 Coordination and Composition

We position associative broadcast with respect to naming and communication models, before describing the specific instance of associative interactions that we have implemented as a coordination and composition model.

4.4.1 Naming Models and Communication Models

Bayerdorffer [11] has developed a lattice taxonomy of naming models and characterized the models of communication which can be implemented in a given model of names. The lattice taxonomy is defined with properties of name models as axes. It is shown in [11] that direct implementation of fully distributed, fully symmetric, minimal communication algorithms among dynamic set of services requires a model of names at the top of the given lattice taxonomy.

Associative broadcast is at the top of the lattice taxonomy of naming systems [11, 12] and thus supports implementation of fully symmetric and fully distributed algorithms for managing membership in dynamic sets. Linda [57] and its derivatives are the only other implemented communication mechanisms known to us corresponding to the highest point of the name lattice taxonomy.

4.4.2 A Coordination-Oriented Implementation of Associative Interactions

The associative interfaces for composition and coordination defined and described in this chapter extend associative *communication* to associative *interaction*. An interaction has the dictionary definition of a mutually agreed upon action conducted by two or more parties. An associative interface for implementation of coordination and composition has two elements: an *accepts* interface and a *requires* interface. Accepts and requires interfaces are extensions of associative broadcast to include specifications for the actions which should result from a successful match of a profile.

An *accepts* interface specifies the set of interactions in which a component is willing to participate. The accepts interface for a component is a set of three-tuples

(*profile*, *transaction*, *protocol*). Multiple members of the set may be associated with each instance of a component model and new instances instantiated at runtime.

A *transaction* [61] specifies the type, functionality and parameters of a unit of work to be executed. A transaction may have an identity and a state which may persist across execution sites. The arguments of a transaction are typed in the invocations as well as in the declaration. In the present implementation, only simple transactions are supported. Complex transactions that persist across multiple component interactions are planned for future work.

The *requires* interface specifies the set of interactions which a component must initiate to perform its intended functions: either its own goals, or what is required to complete transactions that it accepts. The requires interface is a set of three-tuples (*selector*, *transaction*, *protocol*). Multiple members of the set may be associated with each instance of an object and new instances instantiated at runtime.

4.5 Extended Associative Broadcast Coordination and Programming Model

The previous associative broadcast coordination model has been extended into a programming model which enables direct representation of complex interactions with retention of separation of concerns. This model incorporates two additional features: complex conditions for enabling execution of a component and replication for both representation of SPMD parallelism and fault-tolerance.

The conditions for executing and action of a component commonly include receipt of multiple messages. To maintain separation of concerns it is necessary to incorporate this requirement into the coordination model[1]. We introduce the concept of a *firing rule* into the coordination model. A firing rule is a specification of the set of messages which must be received to initiate any action of a component. Additionally, since components may and often will have persistent state, there may be precedence relations among possible enabling message sequences. These extensions are accomplished by adding types to messages and incorporating a conditional

[1]In the previous coordination model, if multiple messages were required to enable an action by a component, the set of actions of the component had to include aggregation of these messages in effect breaking separation of concerns.

expression over message types and local state into the associative interface.

The definition of firing rules used in the extended coordination model is taken from a data flow programming model [97], where rather than waiting on a single input, a node in a data flow graph waits on multiple inputs, possibly in a particular order, before becoming enabled for execution. Firing rules are specified with a Java-like logical syntax. Specifying reception of *either* of two message types R, S is done with a rule $R \mid\mid S$. Reception of *both* of two message types is specified with a rule $R \;\&\&\; S$. Reception of R *followed by* S is specified with a rule $R < S$. These rules can be compounded and grouped with parentheses, such as $(R < S) \mid\mid (R < T)$. The $<$ operator has the lowest precedence, followed by $||$, and $\&\&$ has the highest precedence.

Replication is another feature that must be included in associative interaction specifications to enable facile specification of parallelism and fault-tolerance. SPMD parallelism can be readily implemented by replication of components. Replication of functionality for fault-tolerance can be made transparent and synchronization-free after initialization. If an initiating component starts several replicas of a given component to ensure success in an unreliable environment and each of the replicated components responds by associative broadcast, then the initiating component can safely proceed after the first successful result and set its profile to ignore any results from other replicas. A component can be replicated by adding an index attribute to its profile and instantiating replicas in conformance to the index range. Once specified, a component can be started an arbitrary number of times. The runtime system will provide unique identifiers in a predictable way when launching components so replicas can alter their behavior, or they can all execute in the same way, depending on the needs of the application.

A component in the extended model is a 5-tuple (S, S_0, P, A, R) where S is the state machine which implements the rules for the protocol specification, and the rules for profile and interface changes, S_0 is the initial state, P is the profile of attributes and attribute-value pairs, A is the list of accepted transactions (T, T_A) where T is a firing rule and T_A is the argument signature, and R is the list of requested transactions (T, T_A) where T is a transaction type and T_A is the argument signature. Section 4.7 illustrates the concepts in the extended model in CoorSet language examples.

4.6 Algorithm Formulation

Most distributed algorithms explicitly or implicitly are formulated on the assumption of central control. Coordination models, on the other hand, do not assume central control. Development of algorithms and computations in coordination models therefore requires a shift in development paradigm. Use of a coordination model based on broadcast communication induces a further shift in development paradigm since most distributed computations and coordination models are based on point to point communication. There has been relatively little research in formulation of distributed/parallel algorithms in broadcast models of computation [47].

The development paradigm for distributed algorithms formulated in associative interactions is the integration of component composition and component interactions. An algorithm is specified as coordination among a set of components. Composition defines the structural relationships among components while coordination specifies the behavior of the composed system. Associative interactions use the same mechanism to specify both coordination and composition.

A coordination system implementing an algorithm or computation is specified in terms of a set of attributes in which the profiles and selectors are specified, a set of components from which the algorithm or computation can be composed, a set of protocols in which interactions are specified including message types, the selectors to accompany each instance of a message type, the allowed sequences of messages and the responses to each instance of a message type which is received, and a state machine which implements the coordination protocols which are interfaced to each process or component.

4.7 CoorSet Interface Definition Language

CoorSet has an interface definition language for specification of the behaviors of components in terms of associative interactions. The CoorSet compiler generates Java code to implement the coordination models for each component and a "main" component that starts the application in the runtime system described in Section 4.8. In the example that follows, components of the language that deal with details not directly related to the interface structure have been omitted for clarity; for complete details of the language, see appendix A.

```
component 5 {
  profile ("ReaderWriter", ("EID", 2),
           ("Status", "initializing"))

  execute startUp

  accepts "RequestToRead" processRead ()
  accepts "RequestToWrite" processWrite (Object)

  rule "update_value < (update_done || Collision)" processUpdate

  requests "ReplyFromData" sendReply "Client" (Object)
  requests "update_value" attemptUpdate "ReaderWriter"
     (Object, Integer)
  requests "update_done" completeUpdate "ReaderWriter" (Integer)
  requests "Collision" updateCollide "ReaderWriter" (Integer)
}
```

Figure 4.1: CoorSet definition of a replicated data object store

4.7.1 Readers/Writers Algorithm in CoorSet

To demonstrate the language for describing components, we introduce a generalized distributed readers/writers system implemented in CoorSet. The data objects are replicated for fault-tolerance. Consistency is maintained across non-malicious failures of components and/or runtime creation of additional replicas. This generalized readers/writers problem is rather complex when programmed in conventional distributed/parallel programming languages but is quite simple in CoorSet. The readers/writers system consists of a set of reader/writer objects which store a data item that is replicated across multiple independent stores, and a set of client components which randomly invoke reads and writes of randomly selected data items. Each reader/writer is a single component in the system. Each encapsulates and stores a single replica of a single data item, provides reading and writing facilities to clients, and implements a coordination protocol amongst all the other replicas of that data item when a write is requested. Each replica's profile contains the unique identifier of the data object, and an index to indicate which replica it is.

Each reader/writer component keeps track of the version number of its data item, increasing it each time an update is made. When two updates are attempted simultaneously, meaning they are sent out with the same sequence number, they are said to "collide." When they do, they are aborted. Each then executes an exponential backoff algorithm before attempting the update again with a new sequence number. A definition fragment for the reader/writer component is given in Figure 4.1.

The "5" immediately following the "component" declaration specifies that five instances will be started of a component which has an initial profile of three entries. First, an ID attribute named ReaderWriter distinguishes it from other kinds of components in the system, such as clients. Second, a valued attribute EID (for entry identifier) indicates which data item this object contains. Here the entry identifier is declared in the configuration file, implying that each data object's replicas are declared separately. Third, a valued attribute named Status with the value "initializing." This is the state of the component when it initially comes online, to show that it is not yet operational, and needs to synchronize with whatever other data stores are already in operation. This attribute later changes to values local processing, reading, and writing to reflect the various states it is in when processing requests.

This component type accepts two message types, RequestToRead and RequestToWrite. These requests are made by clients who want to read and write the data item, respectively. In each case, reception of these messages causes execution of the methods processRead and processWrite on the programmer-supplied computational code (not shown), each of which takes the given parameter types.

The readers/writers component also implements a firing rule which first receives an update_value message, and then an update_done message to indicate the update is successful, or a collision to indicate two writes were attempted at the same time and collided.

This component also requests four message types. ReplyFromData is the response sent to clients in response to a request to read or write. It carries a single Object parameter, which will contain a copy of the data item when read. Its default selector of "Client" will be received by all clients, but when such a response is actually sent, the selector will be refined to target only the requesting client. update_value, update_done, and Collision are all sent during the various stages of consensus to attempt an update of the data item, to signal the update is successful,

or conversely, when two attempted updates collide and must be aborted. Each takes the new sequence number of the updated data, and for update_value, the new data value. Each of these are by default targeted to all other data stores by the default selector "ReaderWriter". The optional execute line specifies a method on the programmer-supplied class to execute immediately upon component start-up; if this line is absent, the component just waits for incoming transactions upon starting.

For a simple performance study the data objects were replicated 2 and 4 times. The number of processes reading and writing was varied up to 64, each on a separate workstation on a network. The average number of messages was about $N \times 2.5$ where N is the number of *data object* replicas. Note that the performance of the algorithm is almost independent of the number of readers and writers.

4.7.2 Data Fitting Example

We now present a more complex example of distributed data fitting, motivated by the concept of "greedy reuse" [90]. "Greedy reuse" uses execution of multiple, perhaps redundant components, to ensure the success of a computation by simultaneously executing multiple implementations of a required functionality when it is not certain which implementation should be used. "Greedy reuse" is complex to program in conventional distributed programming systems but simple as a coordination language program. Consider an application that collects a set of data points, and requires approximating them by a curve. There are many possible approaches to data fitting. Consider for illustration a case where it is unclear simply from the data set what method will yield a fit with certain properties required by the application. Possible properties are a minimum of error, compactness of representation, and smoothness of curve. It may be that the requirement is satisfied only by a composition of fits.

Using associatively-coordinated components, several data fits can be executed simultaneously by addressing a data set with a selector that matches to true for the profiles of all data fitting components. The selector can be made more specific if only certain types of fits are desired. This application has more obvious connections between components as is common in more typical coordination models, and the component which initiates the computation is separate from the one that receives the result, to give a linear data flow as illustrated by Figure 4.2. There is

no explicit link between these components, and each circle in fact represents any number of components of that type which may be operating when the request is made. These links should be seen as dynamic, existing only as long as they are required.

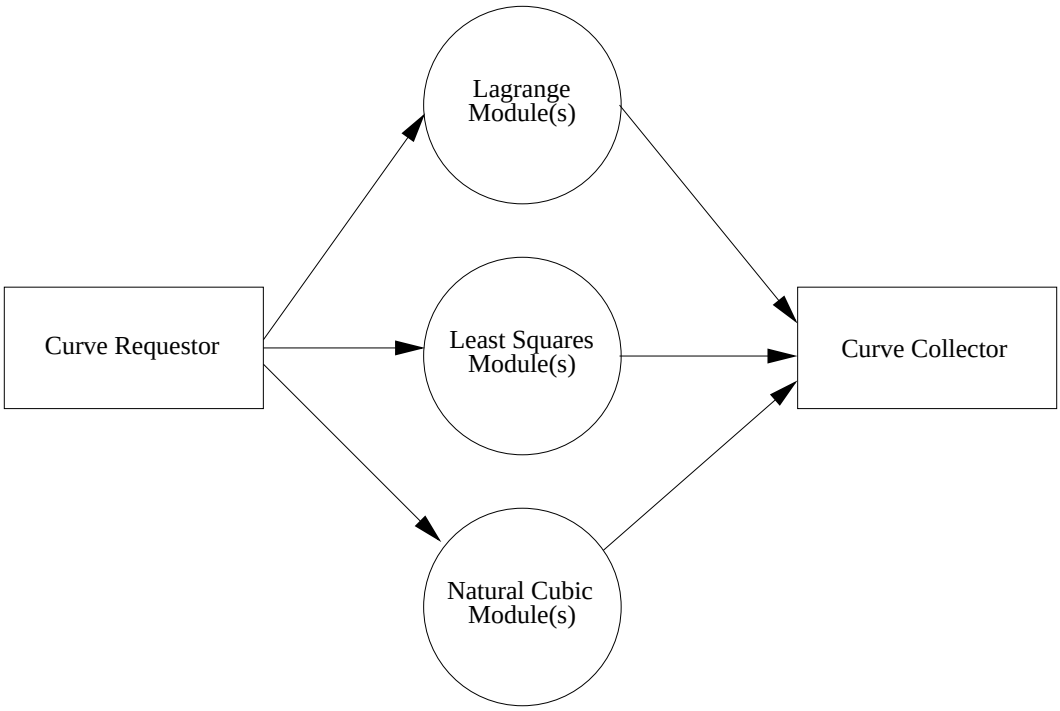

Figure 4.2: Data flow between components of the data fitter application

Transparent replication and fault-tolerance is obtained by having several copies of the same type of component running, and when the calling component receives all of the results, it can compare them to choose the ones which meet the requirements, or alternatively, those which are faulty. In an unreliable environment the initiating component might choose to simultaneously execute multiple copies of another component just to insure that a result is computed and successfully received with high probability.

We have, for this illustration of concepts, implemented components that provide an exact Lagrange interpolating polynomial fit, a least squares approximation, and a natural cubic spline fit. Each component maintains a profile that identifies

it as a data fitting component for the purposes of addressing, as well as profile entries that allow it to be addressed more directly when an application wants only a particular kind of fitting.

The dynamic structure of associative broadcast allows an application to link to all components available at the time a request is made, and to do so without explicit knowledge of what components are available; simply the knowledge of the accepts interfaces used by data fitting components is sufficient.

There are two possible system configurations for the components. The components can be active as daemons on hosts in the broadcast network in which case the initiating component is invoked on some hosts and discovery, linking and execution proceeds as previously described. Alternatively, the components can be in a file with the initiating component or in a library. In these latter cases the runtime system will distribute the components to hosts in the broadcast network and start the associative interaction runtime system.

Figure 4.3 contains the definition fragments for each of the types of components in CoorSet. These interface definitions are used to create the initial configuration of the component network. Some details have been omitted due to space constraints.

In this example, we have five types of components. In this case, three of each of the fitter components is started, as indicated by the "3" after the "component" keyword, to create replicated instances. There is only a single instance of the other components, CurveRequestor and CurveCollector. The types of components are:

- *CurveRequestor*: A component that has collected some data set, and requires it be fit to a curve. It does not accept any transactions, but makes a "DataFit" request to all data fitting components by way of its selector.

- *LagrangeModule*: An exact Lagrange interpolating polynomial. Its initial profile has one attribute called "DataFitter" to indicate that it is a data fitting component, and a valued attribute called "Method" with a value of "Lagrange" to specify the particular kind of data fitter it is. It accepts the "FitData" request, and makes a "FitDataRespnse_Poly" request to send its result.

- *LeastSquaresModule*: A least squares polynomial fitter. Its interface is almost identical to that of the Lagrange module, except that its profile reflects its

being a Least Squares fitter, rather than a Lagrange interpolating polynomial fitter.

- *NatCubicModule*: A natural cubic spline fitter. It accepts the same "FitData" transaction, but responds with a "FitDataResponse_Spline" transaction, which contains a spline rather than a single polynomial.

- *CurveCollector*: A component that accepts the resulting curve fits from the above components. Its default profile contains an attribute called "CurveCollector," which also is the default for selectors for the responses from the above components.

There are also three data types of requests for service:

- *FitData*: A request for a data fit. This transaction has four parameters: the first, a String, specifies a transaction identifier, so that multiple fits may be requested and the responses can be connected with the appropriate request. The next two parameters are arrays of Double values, representing the X and Y coordinates of the data points. The final Integer parameter specifies the maximum order of the polynomial, for polynomial fitters that can bound the polynomial degree.

- *FitDataResponse_Poly*: A response to a data fitting request, containing a polynomial fitting to the data. It carries a String with the transaction identifier for which this is a fit, and an array of Double values representing the polynomial coefficients.

- *FitDataResponse_Spline*:A response to a data fitting request, containing a natural cubic spline fitting to the data. It also carries a String transaction identifier, as well as two arrays of Cubic polynomials. The first is a piecewise parameterized representation of the X coordinates of the spline, and the second is a piecewise parameterized representation of the Y coordinates.

4.7.3 Distributed Computation of Google PageRanks

The Google PageRank algorithm [100] is the computation of the eigenvectors of the lowest eigenvalue of a matrix defined by the link structure of web pages. This

```
component 1 {
  class CurveRequestor
  execute start
  requests "FitData" Request "DataFitter"
     (String, Double[], Double[], Integer)
}
component 3 {
  class LagrangeModule
  profile ("DataFitter", ("Method", "Lagrange"))
  accepts "FitData" processRequest
     (String, Double[], Double[], Integer)
  requests "FitDataResponse_Poly" sendResponse "CurveCollector"
     (String, Double[])
}
component 3 {
  class LeastSquaresModule
  profile ("DataFitter", ("Method", "LeastSquares"))
  accepts "FitData" processRequest
     (String, Double[], Double[], Integer)
  requests "FitDataResponse_Poly" sendResponse "CurveCollector"
     (String, Double[])
}
component 3 {
  class NatCubicModule
  profile ("DataFitter", ("Method", "NatCubicSpline"))
  accepts "FitData" processRequest
     (String, Double[], Double[], Integer)
  requests "FitDataResponse_Spline" sendResponse "CurveCollector"
     (String, Cubic[], Cubic[])
}
component 1 {
  class CurveCollector
  profile ("CurveCollector")
  accepts "FitDataResponse_Poly" processPoly (String, Double[])
  accepts "FitDataResponse_Spline" processSpline
     (String, Cubic[], Cubic[])
}
```

Figure 4.3: Initial configuration of data fitting in *CoorSet*

computation is readily formulated in asynchronous iteration [35]. A distributed, dynamic computation [113] of pageranks has been implemented in CoorSet. Documents which have URL-like links are distributed across a set of hosts (which model web servers) coupled by a broadcast capability. The pageranks are computed in place on the hosts. Pageranks are incrementally computed as documents are added or deleted. A detailed discussion of the implementation of this algorithm is beyond the scope of this paper. On a data set of 1000 documents running on eight processors on a 100-megabit local area network, an average of 1846 messages were required to converge, with an average running time of 68.6 seconds.

4.8 Runtime Support for Launching

The requirements for experimental research on distributed coordination systems in CoorSet are: implementation of timed reliable asynchronous broadcast, configuration and realization of coordination systems on distributed resources and a runtime system which supports the extended associative interactions coordination model specified in Section 4.5. This section defines and describes the capabilities for these three requirements currently implemented for the CoorSet development environment. A system for supporting experimental research on distributed systems needs the following capabilities:

- **Discovery of available hosts.** A distributed launching system must be able to find out what hosts are available to participate in the experiment.
- **Request authentication.** Any system of this type must ensure that only requests with the proper security credentials are honored.
- **Filesystem independence.** A distributed system should not assume a shared file system. Therefore it is its responsibility to see that binaries are transported to the execution sites when launching.
- **Host independence.** The system should handle the loading and execution of code on a variety of architectures.
- **Dynamic system structure.** Such a system should allow dynamic structuring of experiments, as these experiments will often involve joining and leaving protocols, and fault tolerance.

4.8.1 Component Starting Component

The CoorSet development environment can use the "Component Starting Component" (CSC) [73], an environment for launching Java components to configure and instantiate coordination systems for execution on distributed resources. The CSC is deployed on participating systems in a network. Once installed on a host in a network it can stay resident indefinitely. A coordination system is initialized by multicasting a solicitation for available hosts to discover CSCs without *a priori* knowledge of their locations. After receiving service offers, the program initializing the coordination system connects to an appropriate set of responding hosts, and then sends Java bytecode data and startup instructions. The CSC loads the component and starts the component in a different thread in its local Java Virtual Machine (JVM). Communication is guarded by cryptographic signatures to prevent unauthorized use. The CSC provides automated support for distributed systems research that does away with the necessity of manually logging into a number of remote workstations to launch the components of a system. The "main program" generated by the CoorSet compiler consists of instructions to a network of participating CSC units to launch the components of the application across available hosts.

Once the CSC is installed on a network connected by timed asynchronous reliable broadcast[2], then distributed coordination systems can be instantiated in minutes or even seconds. This allows a CoorSet program to be launched from a single point. The runtime system assigns a unique index to each component, which allows components of a like type to differentiate themselves. The runtime system assigns these in a predictable manner based on the format of the configuration file. This allows components running the same code to behave differently if so desired by choosing a control path based on that identifier. This identifier can also be used in interactions where a unique identifier is desired, such as for point-to-point communications, or identities in an election, just to name a few.

The Associative Interactions runtime listens to broadcast messages on the underlying network substrate, evaluates selectors, and forwards matching messages upwards to the application. Data flow semantics are now supported with a component of the run-time system that implements firing rules. Firing rule components inherit standard code that listens to messages and waits for the rule to be satisfied.

[2]Although Ethernet LANs theoretically could take arbitrarily long to deliver a broadcast, with a reliability layer the probability of this is sufficiently small to be neglected.

The computational code connected to that rule is then executed.

Compound components inherit code that provides an event dispatcher for the accepts interface, and a standard application programming interface (API) for runtime modification of the profile, accepts and requests interfaces.

4.8.2 Grid MP

CoorSet also supports the use of United Devices' Grid MP [123] as a launching platform for components. The Grid MP platform uses a dedicated scheduling server that accepts job submissions and coordinates available clients. Each client volunteers its idle time by running the MP Agent, which runs in the background, and when a client is idle, requests work from the server. Work units and programs are downloaded into an isolated work space, executed, and any results are returned to the server for later retrieval. The Grid MP system supports multiple client architectures, and ensures that binaries are downloaded and executed only on compatible hosts.

A special binary package that includes both the Java Runtime Environment (JRE) and the CoorSet libraries is loaded onto the Grid MP server for each supported architecture. By using a special set of global directives in the CoorSet configuration file as described in section A.1.3, the main program generated by the CoorSet compiler is constructed to upload and schedule for execution automatically each component as a separate work unit. The Grid MP server employs caching, which allows the relatively large binary package containing the JRE and libraries to be downloaded once, and cached on client sites to be reused when new components are downloaded and executed. At present, this package must be manually loaded onto the Grid MP server, and its Globally Unique Identifier (GUID) must then be manually programmed into the CoorSet library. This must be done only once for each Grid MP system, which can then be used for any number of CoorSet executions.

Because the clients on a Grid MP system are unlikely to be in a single multicast domain, CoorSet also supports a broadcast emulation over unicast. The software includes a *directory server*, which is started manually in a known location, and whose location is provided as part of the communication parameters in the CoorSet configuration file. The directory server must be manually launched, but can remain

resident for multiple executions. It is common to use the host from which the application is launched as the directory server as well. This server maintains a list of all the profiles of components in the system. When an associatively-addressed message is sent, the directory server examines the selector, constructs a list of matching profiles, and returns a list of unicast endpoints to the sender where the actual message is then sent. There are two limitations of this approach:

- The directory server represents a single point of failure and a scalability-limiting point. The directory server must store the entire profile for each active component in the system. If the directory server fails, the system ceases to function.

- As components update their profiles, they send updates to the directory server. There are two possible race conditions that can occur due to requests and profile updates arriving at different times:

    1. A request is matched against a profile on a component which, due to a profile change, should no longer match that request, but the relevant update has not yet been received. The CoorSet system copes with this condition by checking the selector again at the recipient site, and silently ignoring selectors that no longer match. In this way, the component behaves just as it does in the multicast setting where receiving all messages is commonplace (though, in the case of unicast, highly inefficient), and the directory server is merely an optimization.
    2. A request does not match against a profile on a component which, due to a profile change, should now match. This case is equivalent to a component experiencing a delay that causes its profile not to change to match the request until after the request is made. Although this may lead to less efficient operation of the system, it does not cause incorrect operation of the system, as any application in this model must be designed to tolerate such circumstances, including retrying a request if no or an insufficient number of matching components are found on the first attempt.

4.9 Implementation

The CoorSet language compiler, runtime system, and Component Starting Component are implemented in Java. CoorSet executes either with an implementation based on the Light-weight Reliable Multicast Protocol [84] that operates on top of IP Multicast, or Scribe [109], a multicast overlay that runs on top of the peer-to-peer Pastry [107] protocol. The latter implementation allows for implementation over wide area distributed systems. It can also use a custom broadcast-over-unicast emulation system through the use of the Directory Server, as described in section 4.8.2. The Associative Interface, a class which mediates communication between the application and the network, listens on the multicast socket and evaluates the selectors of incoming broadcasts as the application invokes the message reception API. Matching messages are delivered, and the rest are discarded.

The CoorSet compiler generates Java classes for each defined component type, and a main program invoked to start the system. These generated classes use methods provided by the programmer for the computational part of the component as well as the CoorSet library.

The Component Starting Component is also written entirely in Java. When it receives components to launch, they are launched in independent threads in the same virtual machine. The Java Cryptographic Extensions (JCE), now a standard part of the Java SDK, provide the cryptographic primitives for secure key generation, signature generation, and signature verification for code bundles.

The broadcast model of communication allows greater efficiency of communication on systems like Ethernet where broadcast is the norm, requiring small numbers of messages to reach large numbers of recipients. In the data fitting example in section 4.7.2, a single message is all that is required to request processing by all available fitter units, instead of dispatching a separate message to each one. This represents a savings when some or all of the units are in the same broadcast domain, and invokes all units that are available, allowing for transparent replication of fitter units. The invoker can compare results for consistency to guard against faulty units, and choose the one that best satisfies a chosen metric amongst the various kinds of fits available.

4.10 Related Research

4.10.1 Associative Broadcast

Splice [25] is a coordination model and language which directly makes use of broadcast. It is enlightening to position associative broadcast coordination in the context of a survey of coordination models and languages by Papadopoulos and Arbab [101]. Papadopoulos and Arbab suggest that models of coordination can be approximately classified as data-driven or control driven. Data-driven coordination models are structured by Shared Dataspaces, usually implemented as tuple spaces. Processes or components coordinate by putting or getting data from the Shared Dataspace. Linda [57] and its derivatives and enhancements are the prototypical data-driven coordination models and languages. Papadopoulos and Arbab [101] define control-driven coordination languages as follows.

> In control-driven or process-oriented coordination languages, the coordinated framework evolves by means of observing state changes in processes and, possibly, broadcast of events. Contrary to the case of the data-driven family where coordinators directly handle and examine data values, here processes (whether coordinating or computational ones) are treated as black boxes; data handled within a process is of no concern to the environment of the process. Processes communicate with their environment by means of clearly defined interfaces, usually referred to as input or output ports. Producer-consumer relationships are formed by means of setting up stream or channel connections between output ports of producers and input ports of consumers. By nature, these connections are point-to-point, although limited broadcasting functionality is usually allowed by forming 1-n relationships between a producer and n consumers and vice versa.

Coordination formulated under associative interactions does not, at first glance, classify cleanly as data-driven or control-driven. There is no shared dataspace and while there are messages there are neither ports nor channels. Analogies can be made with both the shared dataspace (tuple space) based coordination models and the control-driven coordination models. Associative interactions can be classified as a hybrid of both approaches.

Other than Splice, the coordination models and languages which seem most similar include those which provide for dynamic creation of processes [9], those based on distributed tuple spaces [108], and those concerned with mobile agents [8].

4.10.2 Runtime Systems

Runtime implementations of coordination models and peer to peer systems are essential for experimental research on distributed implementations of coordination systems. The most closely related research to the runtime system described herein is Klava [19], which implements the Linda [57] model on top of an infrastructure that supports mobile code in a distributed tuple space, including a facility for transporting Java code across a network and starting it in a remote location.

Picco and Buschini [103] describe Linda in a Mobile Environment (LIME) that uses the Linda tuple-space model, dividing the tuple-space amongst a number of mobile agents. They extend the model by allowing the tuple space to contain classes, and using it as the code basis for a class loading mechanism, instead of the local disk.

SPACETUB [120] is a simulation environment for Linda-style languages, as opposed to an actual production environment. Each language is modeled in UML, and interpreted by the modeled class whose methods are the primitives of the language under consideration. Although intended to evaluate Linda-like languages, SPACETUB itself could be used as a coordination language by agents directly invoking SPACETUB's primitives.

Peer-to-peer networks can be viewed as a special case of a coordination model, where coordination is accomplished entirely on a set of agreed upon protocols for interaction. In this way each has a well-defined interface and a method for interacting with peers to request and provide services. The associative broadcast coordination model can be viewed as a peer to peer system with protocols for discovery of services and remote procedure calls. In peer-to-peer and associatively coordinated systems, connections are ephemeral, and exist only so long as two components are actually interacting. Although coordination models are commonly more structured than this, those particular models are part of a subclass of all coordination models. In general, a coordination model makes no such assumptions of communications medium, or the nature of connections between components.

4.10.3 Coordination Languages

HOBS [99], a higher-order calculus for broadcasting systems, models many of the important features of the bare Ethernet. It gives a calculus for reasoning about broadcast systems, including using "filters" on incoming broadcasts. This system has been implemented in the ML-descended language OCaml.

PiLar [40] is an architecture description language that uses the π-calculus to describe its semantics. Each component is also abstracted with a series of exported ports which can be connected to other ports in a strictly point-to-point fashion. PiLar was originally created to describe software architectures, but it is shown that it can be used as a coordination language, and an example of implementation of the Linda [57] model is given.

Manifold [9] is a language which collects groups of components into *manifolds* and encapsulates them into an independent process with its own virtual processor, having its own set of external ports and interconnections amongst the encapsulated members, and reactions to events and other changes in state of the members. Components and manifolds are connected explicitly point-to-point, and once constructed, the system remains static.

Law-Governed Linda (LGL) [89] extends Linda by introducing a *controller* for each entity in the Linda network which mediates communication between the entity and the shared tuplespace. It enforces a set of rules called the law of the system on how entities read and write tuples. These rules are expressed in a predicate calculus such as Prolog.

Coordination Contracts [5] express relationships between objects in a business model. A *contract* between a number of *partners* represents an agreement that certain invariants will always be maintained, and that actions of partners will be coordinated with local actions. These actions are in the language itself, specifying what an object will do when a guard condition is satisfied.

CoML [20] is an XML-based language that describes the interconnections between a group of components implemented in a general-purpose language, for components that operate in an interconnection platform such as CORBA, JavaBeans, or .NET. It uses an event-driven model for communication, where there are event sources that fire events when conditions are met, such as during state changes, event sinks that react upon them, and event data. Components are composed by describ-

ing their interfaces and explicit connections to other components. Connections are changeable during runtime.

Linear Objects [6] are an integration of logic and object programming where the "facts" in a knowledge base include methods defined by classes. Program clauses may have multiple "heads" including references to these methods connected by an operator closely related to logical disjunction. Generation of a search tree creates references to the method signatures. Each step in generation of a search tree corresponds to a restricted form of broadcasting an associatively addressed message.

A CoorSet program is equivalent to a parallel production rule program [128] where both the rules and the object store are distributed. There are two object types: a rule object type has some member variables and three methods, a condition evaluation method, a conflict resolution method and an action execution method, a data object type has some member variables and two methods, an access method and a distribution method.

Web Services Flow Language (WSFL) [82] describes interactions amongst web services in either a flow model, which illustrates a particular business process, or a global model which describes how a set of web services interact without regard to a particular application, but this language is geared specifically towards web services specified in the Web Services Description Language (WSDL). Explicit connections are made between service instances, and using web service interfaces. Grid services [59] address the same idea as web services, but in the context of the computational grid. Services here also use WSDL, but extend it to allow stateful services, discovery, and use of the standard authorization mechanisms present in a grid.

WSFL's successor, Business Process Execution Language for Web Services (BPEL4WS) [7] describes relationships between business entities which use web services for all interaction. It also allows the separate specification of public protocols from private, internal protocols, further underscoring the need for components to be viewed as black boxes with a well-defined observable behavior, irrespective of how the internals work. This allows internal processes to be modified as needed, while still maintaining the same public behavior and protocols. It abstracts web services-style interactions into "partner links."

4.11 Conclusions

The CoorSet development environment provides a capability for readily constructing applications in the extended associative interactions coordination model. CoorSet is a framework for supporting computation in the decentralized, distributed networks studied in this dissertation. It has no built-in mechanism for access control or security, however, and automatically assumes that all other components in the system are cooperative and should be allowed unfettered access to all other components.

Now that has been introduced, we now take the lessons from the taxonomy in chapter 3 to design and implement a standard access control mechanism that can scale to the arbitrary sizes these networks can reach in the next chapter.

Chapter 5

Contractually-Limited Capabilities and CapCoorSet

5.1 Introduction

CoorSet is an example of a distributed computation environment with decentralized control that follows the model given in chapter 2. It provides no access control to operations provided by services, meaning that any access control must be manually implemented by the service, much in the same way access control is done for web services [129].

In this chapter we design and implement an access control mechanism that occupies the portion of the taxonomy from chapter 3 that allows for some control over delegation to be given to the clients, and also allows for state to be distributed to the clients to minimize the burden on the service in the face of arbitrarily many clients.

We formulate access control for distributed service-oriented systems in terms of *contractually limited capabilities* (CLCs). This formulation provides local autonomy[1] and the ability to delegate authorizations to enable composition of applications, yet still allows services to define and enforce limitations on the way capabilities can be used and is scalable with network size. Capabilities and contracts are both well-studied concepts. The implementation reported here is the first time they have been integrated for the purpose of access control to the best of our

[1]The services in a network may have agreed conventions or standards for contract terms.

knowledge. We have implemented this access control system atop CoorSet, a previously reported distributed computation middleware [74], which provides a compiler and runtime system for creating applications by compositions of such services. Services are specified in Java while the interactions among services are specified in the CoorSet coordination language. A compiler generates the communication code with links into programmer-provided computational code in Java. The approach is illustrated with a commonly-used example of web services composition, a stock trading system. In this system, a bank, customers, stock brokers, and stock issuers with a variety of trust relationships use capabilities to manage access to services. We demonstrate how, in the course of a customer purchasing some stock, a stock trading scenario takes advantage of the local autonomy, contractual limitations on capabilities, handshaking to acquire initial capabilities, and delegation provided by the system.

Nagaraj [93] summarizes the arguments given by Blaze et. al. [23] and Tally et. al. [117] that access control lists are an inadequate basis for implementation of access control in distributed systems. The arguments include: the cost of authentication in ACL based access control for distributed systems, and the requirement for delegation and the requirement for locally specified policies for access and trust. This motivates the use of other mechanisms for access control that are less costly, and more easily adapt to the dynamic and uncertain environment of decentralized systems.

The remainder of the paper is structured as follows. We present our approach in section 5.2, including a review of capabilities and the original system on which this model is implemented. We then discuss the properties of this approach in section 5.3, and offer the stock trading example in section 5.4. We then discuss the status of the language and implementation in section 5.5. We discuss other distributed services platforms and systems which have used similar models of access control in section 5.6. Finally, we conclude and consider future work in section 5.7.

5.2 Capability-Based Access Control

Access control systems are typically formulated in three stages: formulation of a *policy*, representation of that policy as a *scheme*, and finally realization of that scheme in an *implementation*. An access control policy is a definition of how a system

should provide or deny access. An access control scheme, as defined by [121], is a state transition system in which access control decisions are specified as changes of state in an appropriate representation such as an access control matrix. A set of access control schemes is an access control *model*. An access control implementation is a realization of a scheme or model for a concrete network of services. The state transitions in a scheme are typically formulated with the assumption of complete and consistent system state. In systems with distributed state, an access control implementation for a distributed system may not be able attain perfect correspondence to a given policy or scheme [75].

This section presents contractually-limited capabilities (CLCs) as an implementation mechanism. Contracts specify the allowed set of state transitions in an access control matrix. The CLC mechanism can be used to implement any precisely defined scheme by appropriate definition of contracts. Section 5.2.1 describes the architecture for an implementation of access control in a distributed network of services. Section 5.2.2 gives an introduction to capabilities, and section 5.2.3 describes our extended, contractually-limited capabilities. Section 5.2.4 defines and describes CLC-based access control as implemented on top of CoorSet, to create the system called CapCoorSet.

5.2.1 System Architecture

Recall that *services* are active entities which provide their own access control, as opposed to passive resources which do not. The services are connected by a network that provides bi-directional communication and a mechanism for discovering other services. The discovery mechanism we describe in section 5.5 can be implemented using a native broadcast or multicast, as is readily available on most local area networks, or using a centralized directory or wide-area multicast where it is not, such as over the Internet.

Each service interacts with other services through an *interface*[2] An interface is a set of *operations*, which are individual units of functionality invoked by clients. In the context of an interaction between two services, we refer to the *client* as the service invoking the operation, and still refer to the service as the service providing the

[2]The conventional definition of an interface is the publicly-exposed portion of a service. In CapCoorSet, the interface defines a number of additional run-time properties as well. These are defined in section 5.2.4.

operation. Human operators are abstracted as services which provide no operations, and so only ever play the part of clients. In our implementation, we split the interface into the interface mediating requests made to the service, as well as requests made by the service, as described in section 5.2.4.

We assume there are cryptographic primitives available for establishing private channels between two services, and also for guarding protected objects against tampering through the use of cryptographic signatures. We describe the primitives used in our implementation in section 5.5.

5.2.2 Fundamentals of Capabilities

A *traditional capability* [43, 114] is a self-validating credential that authorizes access to a resource. It is both a reference to a resource and a set of access rights on it. Possession of a capability implies authorization, and so the access control decision is only validating the capability. For example, in an operating system on a single host, a user holds a set of capabilities to files in the filesystem in a special memory segment, and presents a capability to the kernel in order to execute a desired operation on a file. There are capabilities for reading, writing, deleting, and other file-related operations. These are protected by the kernel from any modification, so when used, no access control list or other information needs to be consulted. These capabilities can then be copied to other users to *delegate* all or part of a user's authority to another. Capabilities are kept in tables, where they are mapped to particular functions or operations, so that validation is mapping it to a valid operation; if no such mapping exists, the capability is by definition invalid. In this way, verification is reduced to function/operation table look-up.

An important property of capabilities is that privileged requests can only be *expressed* when the capability is possessed. Without it, the request cannot even be made. For example, a reference to an object in Java is a simple kind of capability: an object can only be accessed if it has received the reference, through the `new` operator or receiving it as a parameter, but due to Java's prohibition on pointer arithmetic, without the reference no access to the object can even be expressed.

5.2.3 Contractually-Limited Capabilities

We introduced contractually-limited capabilities (CLCs) in chapter 2. We now expand that definition to define the structure of a CLC in the CapCoorSet implementation.

A *contract* is a set of constraints on the use of a capability. It is represented as a (possibly stateful) function defined by the service, associated with an issued capability, and enforced by evaluation of the function by the service at the point of invocation. Input to this function is operation-dependent, but consists of the capability and parameters used in the invocation, together with any state the operation is programmed to use in its decision. The function then returns true or false to indicate whether the invocation should be serviced.

A contract can specify any condition expressible in the programming language in which the contract is written. Contracts are typically simple and require only trivial computational cost. Some common contracts are:

- The trivial contract which always returns true, making it an unrestricted capability,
- A validity period, making invocation permitted only after a certain "effective" date, and no later than a certain "expiration" date,
- A time-of-day-dependent period, making invocation permitted only during certain hours of the day, such as business hours,
- An identity check, binding a capability to a particular identity or set of identities, allowing the implementation of an access control list (ACL),
- A maximum use contract, allowing invocation to occur a pre-defined number of times after which further invocations are denied.

A *contractually-limited capability* is a 6-tuple (I, O, K, C, P, S), where:

- I is a pointer into the operation table of the service component, represented as an integer,
- O is a string containing the human-readable name of the operation as provided by the service,
- K is the certificate of the service,
- C is a contract,
- P is the list of parameter types for the operation, and

- S is the cryptographic signature computed across the other five fields, and signed with the private key corresponding to K.

Presently, the service which offers the capability must be the same as the service which will respond to it. We do not yet support capabilities signed by one service being honored by another, although it is possible to do so.

When invoked, a contractually-limited capability is *satisfied* if and only if:

- I is a valid pointer into the operation table,
- K is the certificate of the service where this capability is invoked,
- The arguments provided are of the correct types as listed by P,
- S is the correct signature as computed across the other five fields, and
- C returns true.

5.2.4 CapCoorSet

We now add CLCs to CoorSet for access control in decentralized systems of services. We call the new system "CapCoorSet" for capability-protected CoorSet. We assume that services in a network are able to communicate bi-directionally when aware of each other's presence, that there exists a discovery mechanism implementing the associative matching described in chapter 2, and that any two services can establish a private channel that cannot be eavesdropped by any other party.

Handshaking and Revocation

Invoking an operation requires a capability, but we have a classic bootstrap problem: how does a client acquire capabilities in the first place? Each service implements the *handshake* operation. We assume that all services possess a capability for the handshake operation on all other services. This is implemented by using a globally-known capability object to invoke the handshake.

Recall that in CoorSet, the requires interface lists all the transactions a component will invoke. In CapCoorSet, we modify this to instead list all the operations that will be acquired through a handshake. Each entry lists the parameter types of the operation required, as well as the selector describing the properties of the service to provide it. A client first discovers compatible services through the associative matching mechanism. The client queries based on the selector for the desired

operation, and discovers a number of compatible services. Assuming the client discovers at least one, it then invokes the handshake capability on any number of those services.

The accepts interface lists all the operations which will be provided by the service, usually in a handshake. Though not shown in the tables in the example in section 5.4, this interface contains mappings to Java code that implements the operation, and also Java code that makes the handshake decision. The handshake operation itself is also always implicitly part of the accepts interface. An invocation of the handshake takes as parameters the name of the desired operation, and supporting evidence to convince the service to issue such a capability. Presently, the only supported evidence is an identity. The application then considers this request, and can then create a capability, with whatever limiting contract may apply, and issue it to the handshaking client.

CapCoorSet adds a third interface, the *capability* interface, which is the internal mapping of capabilities to internal functionality, and is consulted when an invocation is made with a capability. Entries are added to this interface most often by a successful handshake, although application routines may create and issue capabilities as well. The list of parameter types for the mapped operation is also stored. Unlike the first two interfaces, the capability interface only exists at runtime, and is rarely directly modified by the application outside the context of the handshake.

We note that only a pointer and the argument types are stored on the service side; a cryptographic signature providing integrity protection[3] around the CLC allows the client to store it without possibility of modification. In an equivalent implementation using an access control list, these items would have be stored in addition to the identity of the authorized client and the applicable contract. Therefore, even in the case where there is a one-to-one correspondence between capabilities and authorizations, the storage cost to the service side is less with CLCs.

A capability can be revoked for any reason, such as when it is used in a manner inconsistent with its contract. As the first step in validating any capability is looking up its mapping, erasing the mapping revokes that capability. When presented, it will appear as an invalid capability. A client finding his capability invalid, or notified that it has been revoked, can then choose to initiate another

[3]See section 5.5 for the cryptographic mechanisms used.

handshake to attempt to re-acquire a valid capability.

The default behavior is to add an additional entry to the capability table for each capability issued. As a trade-off in favor of reduced space requirements over ability to revoke, the one-to-one correspondence between capabilities and authorizations granted can be relaxed. At the opposite extreme, a single entry can support all capabilities granted to a particular operation; the tradeoff is then that revocation of any capability requires revocation of all capabilities to that operation. It is possible to divide capabilities into a number of groups, with each group using capabilities corresponding to a single entry in the capability table, depending on the needs for space requirements versus granularity of revocation required.

Invocation

An invocation is the submission of a capability to a service, along with any supporting evidence to demonstrate the capability's validity, and arguments for the operation. To prevent eavesdropping, this invocation is sent over a private channel. In most cases, a capability is self-evident and requires no supporting evidence, but in the case where a contract binds the capability to a particular identity, proof of that identity is then required. As we discuss in section 5.5, this is presently the only kind of supporting evidence implemented.

When invoked, the capability and its contract are checked. The cryptographic signature K is verified to ensure the capability's integrity. The pointer I, which is the numerical identifier corresponding to an entry in the capability table, is matched. If both are successful, any restrictions imposed by the attached contract are enforced by evaluating the attached function. This checking may result in recording of state information, as application code is invoked as part of this verification. Should the checking fail, the service may elect to just deny the request, or revoke the capability by removing its pointer from the capability interface. For example, a capability whose contract prohibits its use past a certain expiration date may elect to revoke it as the capability is never usable again.

5.3 Properties

CapCoorSet provides the following properties for decentralized systems of services:

Property 1 *Local autonomy and collaborative behaviors.*

In this setting of decentralized control, each service is its own ultimate authority. This local autonomy gives services the ability to establish standards or conventions for contracts and form cooperative trust relationships, but even allows services to form their own trust hierarchies where access control decisions are given to other services, or executed in cooperation with other services. A completely centralized control structure is then a special case, where the autonomy of all services is voluntarily given to a central decision-maker.

Property 2 *Specification and enforcement of constraints on access.*

Simple capabilities are unrestricted: possessing the capability means its use is authorized in any way the holder sees fit. Associating a contract with a capability provides a means of specifying constraints on the use of capabilities and enforcement of those constraints. HYDRA, for instance, decomposes each access right on a file into individual, atomic capabilities, and so each capability was an unrestricted authorization to perform that particular action. At the service level of abstraction, an operation is rarely so fine-grained, and such unrestricted capabilities do not allow enough control. Instead of encoding such restrictions ad-hoc into the operation itself, CapCoorSet places them into the capability object so they are both evident to the client, and checkable by the service.

Property 3 *Standard mechanism for initial issuance of capabilities.*

We also provide each service with the universal handshake capability that it can always invoke, so each service always has at least one operation it can invoke on any other service. This provides the bootstrap necessary to allow the service to issue capabilities to clients who do not yet have any.

Property 4 *Controlled delegation of authorizations.*

Capabilities can be delegated trivially, just by copying or transferring the capability object to the delegate. This copied capability is only useful insofar as the contract allows, but still allows services to delegate only part of their authority to one another, in accordance with the principle of least privilege. For particularly restrictive policies, however, contracts can be designed to restrict use to a single identity, thus making copying of the capability pointless.

Property 5 *Scalability.*

Validating a capability, except for its contract, is done in near-constant time, based only on local state. At present, the capability interface scales at most linearly in space complexity with the number of capabilities issued. Storing as a hash table gives the near-constant time complexity of that data structure. The contract is implemented as a program defined by the service, and is either stored with the capability, or stored at the service and only the specific inputs to the contract are stored with the capability, and is protected from modification through integrity protection. Programs implementing contracts could in principle be complex and require large amounts of computation and/or network communication, but most contracts we have found useful are simple. The size of the implementing programs are measured in hundreds of bytes or fewer, can be stored in the capability itself, and are evaluated in constant time by the service.

As the contract can be an arbitrary piece of code, it can potentially require an arbitrary computation or communication cost. Contracts we have formulated thus far have all required only constant time, and no network communication, and we do not envision useful contracts incurring an unacceptable cost. See section 5.2.3 for examples of these contracts.

5.4 Example: Stock Trading

We now present a simple stock trading system to illustrate the use of contractually-limited capabilities. This example illustrates collaboration and composition of an application from distributed services and also illustrates typical instances of contracts and delegation. The types of services in this system, and their interfaces, are as follows:

Customer A Customer is the initiator of activity in the stock market, and uses liquid funds to purchase stock. It has an empty accepts interface, meaning it provides no operations. It requires three operations: OpenAccount, which opens an account at the bank; CheckRequest, which requests the bank issue a check from its account; and QuoteRequest, which requests a broker solicit a quote for a particular kind of stock. See table 5.1 for parameters and selectors

in the requires interface. Recall that the requires interface only lists operations for which capabilities are acquired through a handshake. The Customer uses a PurchaseAtQuote capability as well, but this capability is not acquired through a handshake but rather delegated to the customer as the result of a QuoteRequest. Therefore, it is not included in the requires interface.

Operation name	Parameters	Selector
OpenAccount		Bank
CheckRequest	Amount, Recipient	Bank
QuoteRequest	Stock name	Broker

Table 5.1: Customer Requires Interface

Stock Broker A stock broker mediates transactions between Customers and Stock Issuers. It accepts two operations: QuoteRequest, as described under the customer; and PurchaseAtQuote, which accepts payment for a number of shares at a given quoted price, and purchases them on the Customer's behalf. See table 5.2 for this interface.

Operation Name	Parameters
QuoteRequest	Stock name
PurchaseAtQuote	Share count, Check

Table 5.2: Stock Broker Accepts Interface

It requires two operations: CheckRequest, which requests the bank issue a check from its account for the purposes of buying stock from an issuer; and QuoteRequest, which solicits a quote from an issuer on behalf of the Customer. In this case, the Broker discovers with a selector specifically targeted to the particular stock requested by the customer; in this example we use the symbol UTCS. See table 5.3 for this interface. As with the Customer, the PurchaseAtQuote and Check capabilities are not included in the requires interface as they are not acquired through a handshake.

Stock Issuer A stock issuer provides stock certificates in exchange for funds. The issuer acquires no capabilities through handshaking, so its requires interface is

Operation name	Parameters	Selector
CheckRequest	Amount, Recipient	Bank
QuoteRequest		Issuer && Symbol == UTCS

Table 5.3: Stock Broker Requires Interface

empty, although like the Broker it does receive Checks in purchase requests. It accepts two operations: QuoteRequest, as described under the broker; and PurchaseAtQuote also as described under the broker. See table 5.4 for this interface.

Operation name	Parameters
QuoteRequest	
PurchaseAtQuote	Share count, Check

Table 5.4: Stock Issuer Accepts Interface

Note that there is a QuoteRequest and PurchaseAtQuote between the Customer and Broker, and also between the Broker and Issuer. The difference between these pairs will be evident in the system's operation.

Bank The bank contains the funding accounts of all the other services. We assume, for this example, that it is trustworthy in maintaining the accounts. It requires no operations, and so its requires interface is empty. It accepts three operations: OpenAccount and CheckRequest, which have both been previously defined; and Check, which is a capability given in response to a CheckRequest, allowing the redeemer to deposit the attached funds to their account. Unlike the other participants, there is only one Bank. See table 5.5 for this interface.

Each customer, stock broker, and stock issuer opens an account with the bank before any interactions take place. For this example, we start off new accounts with $5,000. The associative matching discovery protocol queries with the selector "Bank" to locate the bank, and then the service handshakes for an OpenAccount capability, presenting their identity certificate as supporting evidence. If an account

Operation name	Parameters
OpenAccount	
CheckRequest	Amount, Recipient
Check	

Table 5.5: Bank Accepts Interface

for that identity does not exist, the bank returns the OpenAccount capability with a contract limiting it to one use, and limiting to the holder of that identity. The capability is then invoked, and the account is opened. This then allows the account holder to handshake for a CheckRequest capability, which is used in the workflow.

The detailed workflow of the system can be found in appendix B.2. We list here the examples of handshakes, invocations, and delegations to demonstrate the various uses of capabilities and contracts in this system.

During the primary workflow of the system, customers query to discover brokers that deal in UTCS stock, and handshake to be issued an unrestricted QuoteRequest capability, that lets them elicit the current quote for the stock. This capability can be freely delegated to any other customers. When invoked, a time-limited PurchaseAtQuote capability is issued by the Broker with the current price, which a Customer can then delegate to any other Customer that capability to share the opportunity to buy at that price. Regardless of how or when the capability is further delegated, it is still only useful within its validity period. If the PurchaseAtQuote is invoked, a Check capability that represents the ability to draw funds from the check writer's account is issued to the funds recipient (first the Broker, then the Issuer). The Check carries a contract that allows invocation by only one specific recipient, and a maximum of one time. Therefore, even if it is delegated, it will be of no use to anyone except the intended recipient.

We see here the properties described in section 5.3. Each service decides what rights it extends through the capabilities it creates, and the contractual limitations imposed upon them. We even see some trust relationships built into the system, in that all parties trust the Bank to behave correctly: in a real stock market, this is a consequence of regulatory oversight. We have seen the following kinds of contracts, which are only a few of those possible:

- Unrestricted capabilities, such as QuoteRequest, which have no restrictions on

their use,

- Limited-use capabilities, such as the Check, which can be invoked at most a pre-defined number of times, and
- Expiring capabilities, such as PurchaseAtQuote, which can be invoked only during a specific validity period.

There are several instances of delegation. First, the QuoteRequest and PurchaseAtQuote capabilities can be shared between customers freely. The Check capabilities are issued from the bank to the account holder, and then the account holder delegates the Check to the intended recipient. The PurchaseAtQuote capability of the issuer is delegated by the broker to the customer, though with an intermediate step that corresponds to the 5% commission taken by the broker.

This example is a simplified instance of a stock market. Some non-access control related properties are simplified, such as the arbitrary choice of selling price, and no imposed limit on the amount of shares an issuer can issue. We also have not yet programmed the customers to trade stock with each other, but they could. Some access control related properties are not constrained, such as the number of purchases that can be requested, or the number of shares that can be requested per purchase. A broker or an issuer could restrict the number or the rate of quotes on a particular QuoteRequest capability, or the number of purchases made on a PurchaseAtQuote capability.

5.5 Language, Implementation, and Evaluation

CapCoorSet provides a custom interface definition language and compiler that automatically generates Java classes for services in the system. Application code must be provided as Java classes to which the automatically generated classes are linked. The runtime system is also Java-based and runs on top of a networking layer. Currently, we support a local area reliable multicast based on Light-weight Reliable Multicast Protocol [84] where requests are multicast and matching services respond, or multicast emulated over unicast through the use of a centralized directory. The directory tracks these associative descriptions, computes the recipient set, and returns that set of identifiers to the client, each of which is then contacted directly.

We modify the CoorSet language for CapCoorSet to provide the additional

specifications required for these interactions. Not only does each accepts interface entry require a mapping to Java code to provide the functionality, it requires a mapping to Java code to make the decision during handshaking. We have also added syntactic sugar keywords to make the language more intuitive. The handshake method by convention takes a standard set of parameters, so only the parameter types for the operation itself is required.

In appendix B.1 we give the definition of the Broker from the example in the CapCoorSet language. The definition specifies the initial accepts and requires interfaces, the initial profile, and an optional method to execute when the service begins running. Both interfaces can be modified at run-time. Applications commonly will alter the selectors used for discovering services throughout execution, but can later decide to offer further services, or decide they require further services.

Private communication channels between services are formed by using Transport Layer Security (TLS) [45], which is provided as part of the standard Java 2 API. For now, we use certificates signed by a single Certifying Authority exclusively for the purpose of establishing private channels, and at present do not consider Sybil [48] or man-in-the-middle attacks. This relationship is not used for any other trust decisions. The only supporting evidence implemented at present for the purposes of satisfying contracts and completing handshakes is the proof of identity negotiated as part of an TLS connection.[4] In the future, we will expand the scope of supporting evidence to allow other kinds of documents.

The Digital Signature Algorithm (DSA) keys attached to these certificates are also used for the cryptographic signing of capability objects, using Java 2's cryptography API. We use the "SHA-256 with DSA" [94, 95] signature algorithm. To allow there to be a universally-known handshake capability, we specifically designate integer pointer zero to be the handshake capability, and make an exception in the capability validation code to automatically pass any such capability, so that they can be created by any client for any service.

[4]Although TLS is generally used only to authenticate the server side of the connection, it supports mutual authentication of the client as well, and we employ it in this mode.

5.6 Related Work

The interface-mediated service system model presented here is shared most closely with Web Services [129], though its interface only presents the operations offered by a service, and none that it requires. Web Services are often light-weight processes that are themselves stateless, relying on back-end databases to retain state, and having access control programmed in as part of the computational function. Discovery is either statically bound, or accomplished through a centralized database, such as UDDI [98].

Capabilities were first introduced by Dennis and Van Horn [43] in the context of multiprogrammed computations, and later appear in a number of operating system applications [81]. They are used as the access control mechanism for files in the HYDRA operating system [39]. They appear currently in operating systems such as EROS [115] and CapROS [80] as the protection mechanism for any privileged kernel accesses. Capabilities moved into user space and were protected by cryptography in Amoeba [118], a distributed operating system. CRISIS [13], a wide-area security architecture, uses certificate chains, where the beginning of the chain belongs to a principal explicitly authorized for access to a resource, and then zero or more *transfer certificates* which delegate that authorization to other principals, with each transfer optionally imposing restrictions on use. The presentation of this certificate chain is part of the invocation, in a "capability-like" way. Capabilities have been targets of both criticism [26] and misunderstanding [87]. Capabilities have seen more recent use in the Traffic Validation Architecture [130] to combat Denial of Service attacks on the Internet. Capabilities in TVA support restrictions on traffic volume and validity periods that can be viewed as particular types of contracts.

Contracts, as presently implemented in our system, can be viewed as preconditions for function invocation. Split capabilities [77] use a simpler approach where a capability is annotated with a listing of access rights, to allow a single capability to be used for multiple operations. ICAP [60] incorporates subject identities into capabilities, to enable additional auditing of capability propagation. Contracts like ours are seen in CRISIS as the imposed restrictions on use made in transfer certificates. Contracts also are used in BPEL4WS [7], a business process language for Web Services, in its description of "business processes" which describe requirements on behaviors for both participants in an interaction. A similar requirement exists

in TYCS [58], a component-based system. Policy languages like PolicyMaker [24], KeyNote [22], REFEREE [36], and D1LP [83] use contracts in a specific language to express local policy, and implement a direct translation to executable code. Contracts have also been explored in mobile ad-hoc or peer-to-peer routing [51] as an incentive for services to cooperate.

5.7 Conclusions

The CLC access control mechanism is a natural approach for implementation of access control among distributed systems of services. It enables the required properties of local formulation of access control decisions, delegation to enable composition of applications, low overhead, and scalability. But the decision on whether or not to extend a capability in the first place, and in what circumstances to revoke a capability, is left to the application programmer.

In the next chapter, we give a mechanism that automatically tracks the behavior of other services in a distributed system as an indicator of trustworthiness, and allows querying of known trusted nodes for ratings of unknown nodes to provide for transitive trust. This mechanism can be incorporated into CapCoorSet as the handshake decision function.

Chapter 6

Automated Trust Decisions using Subjective Logic

6.1 Introduction

Cooperation is vital to the functioning of a system with decentralized control. Establishing and maintaining effective cooperation without the central control inherent in traditional networks, where services may choose not to cooperate for their own benefit, is an open research problem. The success of reputation in non-hierarchical human institutions in encouraging cooperation [64] suggests that a system of observed behavior, tracking of reputation, and cooperation contingent on that reputation encourages cooperation and improves the overall functioning of a decentralized system [85, 106]. Although it is impossible to completely prevent free-loading and uncooperative behavior, reputation methods have shown promise for managing them.

In this chapter we explore a mechanism that uses reputation tracking as an incentive to encourage cooperation. It keeps track of interactions with other services, and when a service has shown a history of cooperation, its rating is suitably high to reflect that. This mechanism also quantifies the uncertainty inherent in a decentralized system. This mechanism can then be used in a system like CapCoorSet when deciding whether or not to grant access to a particular operation.

It is intrinsic to these networks that there is uncertainty in the reputation rating of any service by any other service. Therefore it is reasonable to incorporate

uncertainty into the computation of reputations. It is surprising that uncertainty has not seen more use in reputation computations in the past.

Uncertainty can arise when a service joins the network and thus has no history or when its behavior changes over time. New and unknown services to the network have no history of behavior or reputation on which to base an assessment. Established services are therefore left to deal with newcomers in an arbitrary way. Current approaches choose to assign unknown services a default level of trust: some choose to pessimistically assign a minimum trust and require a building of reputation, and some choose to optimistically assign a neutral or positive level of trust and monitor for uncooperative behavior. In each approach, newcomers become immediately equal to established services whose behavior has warranted their status. In the case of changing behavior, adding the second dimension of uncertainty to a reputation rating enables quantification of variation of behavior which uncooperative services may use to mask their lack of cooperation.

This chapter proposes incorporating a measure of uncertainty based on subjective logic [69] into the reputation to reflect the confidence in that reputation. This allows distinguishing between services which have the default level of trust from those who have earned their trust level through an established history, even if those levels of trust are the same. This also allows distinguishing between services which have consistent behavior and those with time-varying behavior. We do this by employing a new type of reputation which introduces a second dimension to the rating to account for this uncertainty. This also allows services to recognize their lack of knowledge about another service, and seek the opinions of others it trusts, so that cooperative services need not build new relationships from scratch, and uncooperative services can be quickly detected and throttled. These ratings are weighted by the reputation of the recommender, so that services which have proven themselves more trustworthy carry greater weight in their recommendations.

The uncertainty-based algorithm has several parameters and the computed reputation and resulting decisions are strongly dependent on these parameters. To evaluate the effectiveness of the uncertainty-based method and to determine the effects of the parameters, we have implemented a simulation infrastructure for decentralized systems using this uncertainty-based scheme. The simulator also implements other mechanisms for computation as a basis for comparison. The simulator is used to determine nearly optimal parameters for each scheme and how well each

scheme a) ensures cooperative services are able to succeed, and b) throttles free-loading services who attempt to consume without contributing to the system.

The simulations show that for typical decentralized systems, where the majority of services are cooperative, the reputation computation that incorporates uncertainty provides superior discrimination between cooperative and uncooperative behaviors in a variety of network situations, while maintaining an acceptable level of success for cooperative services. These initial results are sufficiently promising to motivate continuing study of reputation computations incorporating uncertainty to additional models for uncooperative behavior and a broader range of network behaviors.

6.1.1 System Model

A decentralized, distributed system is a collection of services with connectivity to one another, but (usually) not to any outside networks. We consider two types of services in this system: *cooperative* services and *uncooperative* services. Cooperative services recognize the need for cooperation in these systems, and recognize that incentive is required to promote this cooperation. In contrast, uncooperative services believe the best course of action is to contribute as little as possible to the network and merely consume, in the hopes that this greediness will go undetected and unpunished.

We consider two uncooperative behaviors. First, uncooperative services can refuse to cooperate when requested. They may refuse at all times, or may cooperate with a small probability to attempt to mask their behavior. The algorithm followed by uncooperative services is given in section 6.3.4. Second, we assume uncooperative services are aware of each other. When providing answers to neighborhood queries for trust opinions, this allows collusion amongst uncooperative services in the form of giving maximally positive ratings for each other, while giving maximally negative ratings for all other services.

As established in chapter 2, we exclude the problem of cheap pseudonyms [56] from this work, which allow a service to rapidly assume a new identity, presumably to whitewash [85] its previously negative reputation. For this analysis, we assume each service has a stable identity.

6.2 Related Work

Reputation systems [106] have emerged as the dominant mechanism for ensuring cooperation in ad-hoc and peer-to-peer networks. These networks cannot assume any pre-existing trust or established relationship between any two services, and so the behavior of other services is the only basis on which the trustworthiness of a service can be judged. Ongoing research into reputation systems continues to examine the effect of a variety of policies on the overall function of an ad-hoc network.

CORE [86] is one system for enforcing cooperation in an ad-hoc network. Only positive recommendations are allowed in what it calls *indirect reputation*, which will allow established cooperative nodes to quickly gain the trust of unknown nodes. Negative ratings are not allowed so that uncooperative nodes cannot unfairly advertise negative ratings for cooperative nodes. Reputation information is regularly exchanged in a reputation dissemination phase, and reputations are slowly aged so that inactive nodes gradually return to an unknown state.

CONFIDANT [31] concerns itself only with routing, so there is only one type of functional trust. In contrast to CORE, CONFIDANT monitors only for suspicious events. Whenever bad behavior is experienced, the Trust Manager component of a CONFIDANT node broadcasts an ALARM message to its neighbors. ALARM messages from other nodes are weighted based on the trust level of the alarming node when making decisions about changing local trust values. This causes CONFIDANT to act more quickly and more decisively in response to bad behavior, making it less susceptible to traitors who act cooperatively for a time to build up a good reputation and then turn uncooperative to profit from it, but conversely causes it to treat more harshly cooperative nodes who experience benign failures.

Jøsang, Hayward, and Pope give an access authorization method [68] for use in web services-like systems which use delegation graphs based on the same Subjective Logic used here. Access is allowed to a resource if, through a given chain of subjective logic opinions, the resulting calculated opinion meets a minimum expected value. The mechanism proposed allows arbitrary graphs of delegations of authorizations, but does not address how local opinions are formed, nor does it address changing opinions.

EigenTrust [72] is a system to ensure cooperation in file-sharing networks.

It aggregates trust information from peers by performing a trust computation. In this system, a node has a global reputation value, instead of each node having its own opinion of it. These values are kept in a *trust vector*, that is the computed eigenvector of a matrix made of all local trust values. This is the extreme case of sharing opinions, where the transitive sharing eventually converges to a globally consistent vector, for a large enough network. This scheme seeks to reduce the number of inauthentic downloads, however, and not selfishness of nodes.

Moreton and Twigg [91] give a peer-to-peer routing cooperation enforcement mechanism that also recognizes different kinds of trust, such as trust for providing a service and trust to recommend another to provide a service. They provide a general model for transitive trust networks, and requirements on the combinator operations on trust opinions. They then make reference to an earlier edition of the subjective logic formulas that the mechanism in this chapter uses, though they do not explore the value of uncertainty.

Kraft and Schäfer [78] give a reputation scheme for access to an ad-hoc network that makes use of an earlier version of subjective logic, which accounts for the uncertainty of a rating. This scheme also makes use of weighted neighborhood queries in the case where no local opinion exists, but does not make use of neighborhood queries if some local information is available. Its use of uncertainty is limited to the case where an opinion is entirely uncertain by virtue of being non-existent.

Buchegger and Le Boudec [32] give a similar Bayesian approach to reputation systems as is used in this paper. It makes more conservative use of recommendations to minimize the impact of liars and colluding groups. This scheme maintains its rating as actual beta distribution parameters. This method was evaluated by applying it to CONFIDANT, also by the same authors, which as noted above deals only in negative recommendations.

Ben Salem et. al. [15] apply reputation to the problem of selecting Wireless Internet Service Providers (WISPs) when multiple ones are present. In this case, what has a reputation is the WISP, so that mobile nodes can select a trustworthy one with confidence. A global reputation for each WISP is maintained by a *trusted central authority* (TCA). When a mobile node concludes its interaction with a WISP, it reports the quality of service it received to the TCA, which updates that WISP's reputation accordingly. Future mobile nodes query these reputations when selecting a WISP to use.

6.3 Reputations with Uncertainty

We propose incorporating the uncertainty of an opinion into trust decisions to distinguish new and unknown services from services with an established interaction history and to more effectively detect efforts by uncooperative services to mask their behavior with temporary cooperation.

6.3.1 Subjective Logic

Subjective Logic [69] is a system of rating where each *opinion* is a 4-tuple $\omega_x = (b_x, d_x, u_x, a_x)$ where b_x is the *belief* in a particular statement x, d_x is the *disbelief* in that statement, u_x is the *uncertainty* of the statement, and a_x is the *uncertainty weight* (referred to as the *base rate* in [69]). All of $b_x, d_x, u_x, a_x \in [0.0, 1.0]$, and $b_x + d_x + u_x = 1.0$. This linear constraint restricts possible points to the two-dimensional triangular subspace shown in figure 6.1 [69].

Belief and disbelief are intuitively obvious. The uncertainty of a rating reflects the confidence in the service's knowledge about the statement under consideration; an uncertainty of 1.0 represents a statement about which a service has no basis for any conclusions. The uncertainty weight determines how uncertainty is viewed as belief when the opinion is used, or in other words, how optimistically uncertainty is viewed. When an opinion is used in a decision, it is projected onto the belief/disbelief axis through its *effective trust* $E(\omega_x) = b_x + a_x u_x$. An uncertainty weight of 0.0 causes uncertainty to be viewed as disbelief, and an uncertainty weight of 1.0 causes uncertainty to be viewed as belief. An uncertainty weight of 0.5 causes uncertainty to be viewed half as positively as actual belief. For example, if an opinion is (0.5, 0.0, 0.5, 0.5), meaning that it is half-believed and half-uncertain, its effective trust is $E(\omega_x) = b_x + a_x u_x = 0.5 + (0.5)(0.5) = 0.75$. An entirely uncertain opinion, $(0.0, 0.0, 1.0, a_x)$, will then always have an effective trust equal to the uncertainty weight, as $E(\omega_x) = b_x + a_x u_x = 0.0 + 1.0 a_x = a_x$. The uncertainty weight then becomes the default opinion for unknown services.

6.3.2 Forming Opinions

Subjective Logic gives a framework for expressing opinions about any logical statement. The only logical statement we consider here is the trustworthiness of another

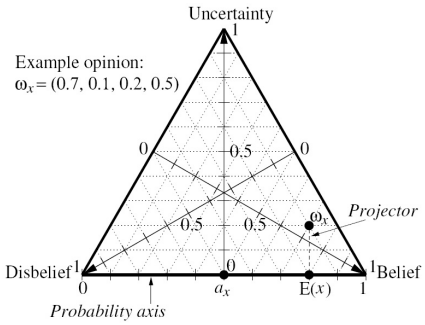

Figure 6.1: Belief/disbelief/uncertainty triangle

service. Although subjective logic gives formulas for combining opinions, it leaves open the question of how opinions are formed in the first place. This suggests that a mechanism must be devised and evaluated experimentally. We propose the following scheme of formulating opinions.

Any unknown service is automatically assigned an entirely uncertain opinion. We will use an uncertainty weight of 0.5, so that unknown services are by default assigned a median level of trust. Therefore, the default opinion is $(0.0, 0.0, 1.0, 0.5)$. Let $\delta \in [0.0, 1.0]$ be a parameter that determines how much a rating changes after an individual interaction between services. We classify an interaction as positive, negative, or uncertain. Each positive or negative interaction increases the rating service's knowledge, and therefore decreases uncertainty. In the following formulas, we omit the subscript x and merely refer to b, d, u, a from each opinion 4-tuple. Opinions are updated through the following formulas:

- After a positive interaction with another service,
  - If $u \geq \delta$, $b := b + \delta$ and $u := u - \delta$.
  - Otherwise, $b := b + \delta$, $d := d - (\delta - u)$, and $u = 0.0$. As uncertainty is exhausted, positive interactions decrease disbelief. Should this cause d to decrease to less than zero, $d := 0.0$ and $b := 1.0$ as this also would cause b to increase past 1.0.

- After a negative interaction with another service,
  - Likewise as for a positive interaction, with the roles of b and d swapped.
- After an uncertain interaction with another service,
  - If both $b, d \geq \delta/2$, $b := b - \delta/2$, $d := d - \delta/2$, and $u := u + \delta$.
  - If only $b < \delta/2$, $b := 0.0$ and $d := d - (\delta - b)$. Still, $u := u + \delta$.
  - If only $d < \delta/2$, $d := 0.0$ and $b := b - (\delta - d)$. Still, $u := u + \delta$.
  - If both $b, d < \delta/2$, then $b := 0.0$, $d := 0.0$, and $u := 1.0$.

These mechanisms embody the philosophy that so long as uncertainty exists, when information becomes available, belief and disbelief increase monotonically. When uncertainty is eliminated, this then becomes a one-dimensional measure between belief and disbelief, unless uncertainty is re-introduced. Uncertainty may be re-introduced by a regular aging of opinions, or by actions detected which are not necessarily uncooperative, but are suspicious. For instance, an action that is decidedly uncooperative can cause a negative interaction, whereas a service failure that could be a transient network failure could be viewed as uncertain. The uncertainty weight allows the severity of these uncertain interactions to be tuned.

6.3.3 Using Cooperation to Overcome Uncertainty

One form of cooperation that can assist the functioning of a network is the sharing of opinions, so that cooperative services need not slowly build trust with everyone, and that uncooperative services can be more rapidly detected. We call these queries *neighborhood queries*. A neighborhood is a set of known services from which recommendations can be elicited. The exact composition of the neighborhood will depend on the particular system used. [69] gives two sets of formulas for combining opinions: one for computing a transitive opinion, and one for computing a consensus opinion, which we will reproduce here.

Let us consider three services c, s, r, where c is a client requesting service from s, and s solicits the opinion of a recommender r in deciding whether to grant the request. s weights this opinion by its opinion of r, and computes the *transitive opinion* $\omega_c^{s:r}$ in equation 6.1. In the following formulas, b_y^x represents the belief in the trust of service x by service y, and similarly for d, u, a.

$$
\begin{aligned}
b_c^{s:r} &= b_r^s b_c^r \\
d_c^{s:r} &= b_r^s d_c^r \\
u_c^{s:r} &= d_c^r + u_r^s + b_r^s u_c^r \\
a_c^{s:r} &= a_c^r
\end{aligned}
\tag{6.1}
$$

Let us now consider three services s, t, c, where s, t are both services, and c is a client about which s and t each have an opinion. To combine these possibly conflicting opinions into a single opinion that reflects both individual opinions in a fair way, we use equation 6.2 which aim to reduce total uncertainty in both opinions in computing the *consensus opinion* $\omega_C^{s \diamond t}$.

$$
\begin{aligned}
b_c^{s \diamond t} &= (b_c^s u_c^t + b_c^t u_c^s)/(u_c^s + u_c^t - u_c^s u_c^t) \\
d_c^{s \diamond t} &= (d_c^s u_c^t + d_c^t u_c^s)/(u_c^s + u_c^t - u_c^s u_c^t) \\
u_c^{s \diamond t} &= (u_c^s u_c^t)/(u_c^s + u_c^t - u_c^s u_c^t) \\
a_c^{s \diamond t} &= a_c^s
\end{aligned}
\tag{6.2}
$$

Where both opinions are entirely certain, and the denominator $(u_c^s + u_c^t - u_c^s u_c^t)$ is zero, we take the limit and compute a simple average of the two opinion values.

These formulas allow us to implement *neighborhood queries*. Let $\beta \in [0.0, 1.0]$ be the *maximum uncertainty* past which a service concludes its local information is lacking, and queries known neighbors for the opinion of a client. For a given service s and client c, if $u_c^s \geq \beta$, s queries for ratings of c. Let R be the set of recommending neighbors who respond with an opinion. We present formulas to compute an aggregate opinion $\omega_c^{s:R} = (b_c^{s:R}, d_c^{s:R}, u_c^{s:R}, a_c^{s:R})$ based on individual weighted opinions in equation 6.3.

$$
\begin{aligned}
b_c^{s:R} &= \frac{\sum_{r \in R} b_c^{s:r}}{\sum_{r \in R} b_r^s} \\
d_c^{s:R} &= \frac{\sum_{r \in R} d_c^{s:r}}{\sum_{r \in R} b_r^s} \\
u_c^{s:R} &= 1.0 - b_c^{s:R} - d_c^{s:R} \\
a_c^{s:R} &= \frac{\sum_{r \in R} a_c^{s:r}}{|R|}
\end{aligned}
\tag{6.3}
$$

[69] assumes that each uncertainty weight a is always the same. We relax this assumption by taking the unweighted average across all uncertainty weights. This opinion, which is the gestalt opinion of the entire network from the point of view of s, can then be combined with ω_c^s through the consensus equation 6.2 to make the

final decision. Note that the local opinion remains unchanged by this process. To avoid excessive query traffic, neighborhood query results can be cached for a short period when a client makes several successive requests.

6.3.4 Protocol Behavior

Each service provides an unspecified operation to any other service. We keep this generic to encapsulate any sort of operation, such as forwarding packets, or providing a request-response service. We assume that when a client makes a request, the service returns a result that indicates success, failure, or timeout. The protocol for for a cooperative service pair s, c is as follows:

1. Client c requests service from service s.
2. s retrieves its opinion ω_c^s from its local storage.
3. If $u_c^s \geq \beta$, s invokes the neighborhood query protocol.
   (a) s broadcasts a request for opinions for c.
   (b) Any service r whose opinion is not entirely uncertain ($u_c^r < 1.0$) responds with its opinion ω_c^r.
   (c) s weights each received opinions using equation 6.1, combines them all into an aggregate opinion using equation 6.3, and combines that opinion with its local opinion using equation 6.2.
4. s computes the effective trust E of either its local opinion if it was sufficiently certain, or of the network opinion if not.
   (a) Let $\alpha \in [0.0, 1.0]$ be the minimum trust required for cooperation. If $E \geq \alpha$, s returns success, representing its providing the desired service. c then records a positive interaction with s.
   (b) If $E < \alpha$, s returns failure, representing its refusal to provide service to a client it deems uncooperative. c then records a negative interaction with s.

The protocol for an uncooperative service s is as follows:

1. Client c requests service from s.

2. Let $\gamma \in [0.0, 1.0]$ be the probability that an uncooperative service will cooperate in an attempt to hide its lack of cooperation. s flips a coin weighted by probability γ.

   (a) If the coin flip indicates to cooperate, s provides the service to c, and c records a positive interaction with s.

   (b) if the coin flip indicates not to cooperate, s refuses service to c, and c records a negative interaction with c.

6.4 Simulation Architecture

Our simulation architecture emulates any number of services in a network. Each service is permitted one action in each iteration of the simulation, where it chooses a random interaction partner, and requests service from that partner. The partner then elects to provide or deny that request, and the service then processes that result with respect to its reputation tracking. For each set of parameters, the simulation is run a configurable number of times (in our experiments, 20) with a different seed to the random number generator that selects interaction partners. These results are then collected, and the mean, standard deviation, minimum, and maximum of each metric is observed.

To ensure a valid comparison between metrics, the sequence of random seeds is the same, to ensure that for a particular run i under various sets of parameters, the interaction partner choices remain the same so that the only variation is the choice of parameters.

Recall that services are one of two possible types: cooperative services, which are interested in the overall functioning of the network and will cooperate with other services it believes to be cooperative, and uncooperative services, which are interested in making use of the network while incurring minimal expense.

The simulation engine supports the following operational parameters:

- The number of services n in the network, as well how many are cooperative and uncooperative,

- The maximum number of steps each run may take before concluding convergence will not likely occur,

- Time intervals during which a particular service is active,
- For each uncooperative service, the time point t_m at which the service begins to act uncooperatively. This allows an uncooperative service to behave cooperatively to earn a positive reputation, before attempting to take advantage of it,
- Whether or not uncooperative services are aware of each others' identities, and will collude with one another in the form of always providing maximally positive recommendations for each other, and maximally negative recommendations for everyone else,
- The probability $\lambda \in [0.0, 1.0]$ that a benign network failure will cause a service request to go unanswered,
- The minimum trust level $\alpha \in [0.0, 1.0]$ a client must have for a cooperative service to cooperate with it,
- The maximum uncertainty $\beta \in [0.0, 1.0]$ of a rating above which a neighborhood query will be employed before making an access control decision,
- The probability $\gamma \in [0.0, 1.0]$ that an uncooperative service, in an attempt to evade detection, will cooperate with a request in the hopes it will allow further access later,
- The reputation mechanism to use (subjective logic table, recent history table, or rate of change),
- For the subjective logic table mechanism, the δ by which the rating changes at each interaction,
- For the recent history and rate of change tables, the maximum history length maintained.

In our experiments, each run continues until the mean effective opinion of all uncooperative services drops below the minimum trust level α. For now, these values are used uniformly across all services. We assume the network has perfect connectivity, modulo the probability of a random benign failure as regulated by the parameter λ. In our experiments, we imposed a maximum of 10,000 iterations per run, and conclude that any run which reaches 10,000 iterations will not converge.

6.5 Performance Analysis

The objective of this analysis is to determine the impact of the subjective logic reputation sharing scheme on the ability of services in a decentralized system to maintain functionality while detecting the actions of uncooperative services. We compare this scheme against two other, simpler methods, casting each into a subjective logic opinion to allow for easy comparison:

- A record of recent history similar to the approach of CORE [86], which tracks the number of positive negative, and uncertain interactions. For the length of history h kept, the number of positive interactions p, the number of negative interactions n, and the number of uncertain interactions u of a service s with a client c is mapped into a subjective logic opinion as follows: $\omega_c^s = (p/h, n/h, u/h, 0.5)$.
- A rate of change, which like the recent history table, tracks the number of each type of interaction for a given history length h. This method tracks how long the latest trend in client c's behavior has been. Formally, for recent history $H = (i_1, i_2, \ldots, i_h)$, the trend t is the length of the suffix $(i_k, \ldots, i_h)$ such that all i are the same type of interaction. This trend is then mapped into a subjective logic opinion as follows:
  - If the trend is positive, $\omega_r^s = (t/h, 0.0, 1.0 - t/h, 0.5)$.
  - If the trend is negative, $\omega_r^s = (0.0, t/h, 1.0 - t/h, 0.5)$.
  - If the trend is uncertain, $\omega_r^s = (0.0, 0.0, 1.0, 0.5)$.

6.5.1 Metrics

We consider the following metrics of performance:

- **Run length to detection.** One factor of interest is how long it takes for cooperative services to detect and throttle uncooperative clients. The run length to detection is the number of iterations at which the mean of the cooperative services' opinions of the uncooperative services is less than the minimum cooperation parameter α, with a standard deviation of no more than 10%. It is desirable for this number to be as small as possible.

- **Percentage of cooperative services' success.** The goal of the reputation computation is rewarding cooperation. The most important metric is the success of cooperative services. We therefore measure the percentage of request attempts made by cooperative services, including those unknowingly made to uncooperative services, which are successfully fulfilled. It is desirable for this percentage to be as high as possible.

- **Percentage of uncooperative services' success.** Also of interest is how successful uncooperative services are, particularly up until the point where total detection occurs, as described above. It is desirable for this percentage to be as low as possible.

6.5.2 Simulation Results

A factorial experiment was run that examined the following sets of parameters:

1. Network size: 10, 20, and 40 services.

2. Probability that an uncooperative service chooses to cooperate: 0%, 10%, and 20%.

3. The maximum uncertainty an opinion can have beyond which a network query is employed to make the final decision: 1.0 (no network queries), 0.8, 0.5, and 0.0 (always do a network query).

4. The probability that a benign failure causes a request or response to be lost: 0%, 5%, 10%, and 20%.

5. For the subjective logic table, the following values for δ were tried: 0.05, 0.1, 0.2, 0.3, 0.4, 0.5, and 1.0.

6. For the recent history and rate of change tables, history lengths of 10 and 100 were tried.

For this experiment, we fixed the uncooperative population as one quarter of the total services. Uncooperative services acted uncooperatively for the entire run of the system, except for random cooperation controlled by parameter γ.

In table 6.1, the mean number of rounds required to converge under the various parameter values for a network of 40 services are given. Also given are the mean percentage of requests made by cooperative services that succeeded up until the point of convergence, and the mean percentage of requests made by uncooperative services that succeeded up until the point of convergence. Due to space constraints, not all data and cases are presented here, but just a few to be sufficiently illustrative. In the table, Method is the reputation tracking mechanism used: SLT for Subjective Logic Table with the chosen δ listed, HIST for recent history with the history length listed, and CHG for the rate of change table with the history length listed. P(Coop) is the probability expressed as a number on the unit circle of whether or not an uncooperative service will randomly choose to cooperate, P(Failure) is the probability a packet will be randomly lost, and Unc is the maximum uncertainty past which a network query is employed. Round is the average iteration at which the convergence condition listed above is reached (or ∞ if convergence was not reached), Coop. % is the average percentage of requests from cooperative services that succeeded, and Uncoop. % is the average percentage of requests from uncooperative services that succeeded up until convergence was achieved.

Time to Convergence

In the most ideal case, where an uncooperative service never cooperates and benign failures do not occur, the rate of change table reached convergence in the smallest number of iterations. The rate of change table benefits from this unchanging behavior, and after just a few negative interactions provides the most negative rating. When the probability of uncooperative service cooperation becomes greater than zero, the rate of change table takes the longest to reach convergence, if it reaches convergence at all: the changing behavior of the uncooperative services upsets the rate of change often enough that the effective rating stays close to the neutral point.

In all cases where the uncooperative service cooperation probability was nonzero, the subjective logic table converged more quickly.

Although the data shown in table 6.1 shows an increase in convergence time for $\delta = 0.1$ over $\delta = 0.05$ for 40-service networks, in 10 and 20-service networks $\delta = 0.1$ resulted in a lower convergence time. Given that more aggressive values of δ resulted in lower convergence time in the 10-service network, this suggests there is

Method	P(Coop)	P(Fail)	Unc	Round	Coop. %	Uncoop. %
SLT (0.05)	0.0	0.0	1.0	343.5	100	81.67
SLT (0.1)				396.2	100	39.22
SLT (0.5)				189.8	100	20.69
SLT (1.0)				164.9	100	23.83
HIST (10)				396.3	100	39.21
CHG (10)				164.9	100	23.83
SLT (0.05)	0.1	0.0	1.0	498.2	100	75.37
SLT (0.1)				573.05	100	34.81
SLT (0.5)				∞	100	11.37
SLT (1.0)				∞	100	10.37
HIST (10)				1209.65	100	20.92
CHG (10)				∞	100	34.22
SLT (0.05)	0.0	0.0	0.5	343.5	100	57.15
SLT (0.1)				396.2	100	25.36
SLT (0.5)				189.8	100	11.87
SLT (1.0)				164.9	100	9.18
HIST (10)				396.3	100	25.35
CHG (10)				164.9	100	9.18
SLT (0.05)	0.0	0.1	1.0	338.85	89.94	74.53
SLT (0.1)				396.15	89.94	35.28
SLT (0.5)				194.6	82.58	18.21
SLT (1.0)				170.8	72.20	20.78
HIST (10)				396.3	89.94	35.26
CHG (10)				170.8	78.96	20.78
SLT (0.05)	0.1	0.1	1.0	533.1	89.92	64.73
SLT (0.1)				547.4	89.96	31.95
SLT (0.5)				∞	42.02	9.20
SLT (1.0)				∞	3.45	8.44
HIST (10)				778.8	74.88	28.25
CHG (10)				∞	89.98	24.46
SLT (0.05)	0.1	0.1	0.5	533.1	89.89	53.40
SLT (0.1)				547.4	89.94	26.30
SLT (0.5)				∞	41.87	9.08
SLT (1.0)				∞	3.43	8.25
HIST (10)				778.8	89.97	20.25
CHG (10)				∞	75.65	1.32

Table 6.1: Run length to total detection in a network of 40 services

a relationship between network size and choice of δ, and that as the network grows, δ must be chosen more conservatively.

In the 10-service network, a δ choice of at least 0.5 improved the time to convergence as well as reduced the uncooperative success percentage, even in the presence of uncooperative service cooperation and benign failures, although there were also reductions in cooperative success percentages as well. In the 20 or 40 service networks, however, these large values for δ caused ratings to thrash by such a large amount that convergence was sometimes not reached in the presence of uncooperative service cooperation and benign failures.

Success Percentage

The subjective logic table with a conservative choice of $\delta \leq 0.1$ maintained a mean successful service percentage of at least 79%, even in the presence of 20% random loss and 20% chance of uncooperative cooperation. This is encouraging, as no reputation mechanism can be useful if legitimate requests are not usually serviced.

A more aggressive δ always resulted in a lower percentage of success for uncooperative services, so another guiding factor in choosing δ is the point past which cooperative services are unable to get an acceptable level of service. Even though convergence was not reached in some cases, high levels of success remained present with a low probability of benign network failures. As this probability increased towards 20%, the resulting unfair punishment of cooperative services caused a large decline in their success rate. This suggests that the aggressiveness of opinion changes quantified by δ must be tempered by the perceived reliability of the network. The more unreliable the network, the more conservatively δ must be chosen.

Even though the recent history and rate of change methods did not converge as often, uncooperative success percentages were comparable or even lower than those for the subjective logic table when only a short history (on the order of 10 entries) was used. Unfortunately, as conditions grew increasingly hostile, in terms of increasing uncooperative service cooperation and benign network failures, cooperative service success declined rapidly. A longer history, such as remembering the last 100 interactions with a particular service, caused the uncooperative success percentage to rise when uncooperative services cooperate randomly. Both of these methods benefit from the harsh treatment of misbehaving services, but in hostile

conditions when cooperative services are unable to cooperate due to benign failures, this same harshness depresses the overall function of the network.

Effect of Neighborhood Queries

Neighborhood queries had no effect on reducing the overall time to convergence, and a negligible effect on the success of cooperative service requests. It had a significant effect on reducing uncooperative service success. For example, in table 6.1 under ideal conditions, uncooperative services succeeded 81.67% of the time under subjective logic with $\delta = 0.05$, and 39.22% of the time with $\delta = 0.1$. These reduced to 57.15% and 25.36% respectively when neighborhood queries were employed when uncertainty was 0.5. This trend held consistently between any two experiments where the only difference was an increase in the use of neighborhood queries. However, the usefulness of these queries lessened as the maximum uncertainty decreased past 0.5, and uncooperative success percentage remained the same from a maximum uncertainty of 0.5 to a maximum uncertainty of 0.0.

6.6 Conclusions

These preliminary studies demonstrate the potential of incorporating uncertainty in reputation computations in the case of systems where the population of cooperative services is high. Using a subjective logic-based system along with using uncertainty as part of the rating to distinguish unknown services from known and untrustworthy services, helps ensure a high level of service for cooperative services while throttling uncooperative ones, even when they cooperate as much as 20% of the time in an attempt to mask their uncoooperative intent. The results indicate that even with no expectation of the behavior of the network, using subjective logic with a conservative value for δ gives good performance. With additional knowledge, such as network size, network failure rate, or likelihood of uncooperative service cooperation, δ can be made more aggressive to optimize the performance further.

A problem with any reputation system where opinions are shared is a denial of service attack in the form of uncooperative services advertising artificially high ratings for its colluders and artificially low ratings for everyone else [131]. By not only weighting recommendations by the trustworthiness of the recommender, as other approaches have done, incorporating uncertainty into this process allows the

effect of recommendations from unknown services to be minimized, or eliminated entirely, as opposed to given partial weight as an opinion from a known but neutral service would receive. This allows the bogus recommendations of uncooperative services to be discarded, and allows those services to be more rapidly throttled, assuming the uncooperative services have not themselves become trusted in the first place. Using this method rather than pessimistically assigning all unknown services a minimum level of trust, which would also discount their recommendations, allows them to receive enough access to participate in the network and therefore the opportunity build up a reputation.

This work is still in its preliminary stages, and requires much more evaluation to justify this method as a general purpose mechanism. A key limitation of our simulations so far is the high population of cooperative services and the relatively simple behavior of uncooperative services. Avenues for further evaluation of this mechanism are given in section 8.1.3.

In the next chapter, we give a logic for proving properties about decentralized systems, that goes beyond using just experimental evaluation and testing to show desirable properties in service-oriented systems.

Chapter 7

Logic for Verification of Access Control Properties

7.1 Introduction

Chapter 5 showed that decentralized, distributed systems can be structured so that decisions on access control can be implemented using mechanisms compatible with decentralized control, and chapter 6 showed experimentally how these decisions can be made where there is uncertainty stemming from a lack of central control. Although thorough testing and simulation can provide a convincing case that a system is correct and behaves as desired, it is rarely possible to exhaustively test all possible executions. In the previous chapter, the implementation was tested with a series of random executions to provide statistical confidence in the methods, as all possible executions could not be feasibly tested.

To obtain confidence in the behavior of such a system, we must verify it more rigorously. This requires an appropriate mathematical logic, such as those which form the basis of software verification methods [102]. Verification logics have been formulated for more traditional distributed systems [1, 33, 34, 41] but these lack certain properties that are required when there is decentralized control. They focus on a set of known and static trust relationships. Their goal is to deduce cryptographic guarantees such as authentication, privacy, or message integrity. They may show that the transitive closure of a set of trust relationships can be calculated, to prove two otherwise unaffiliated services have a basis for secure interaction.

Recall from chapter 2 that our adversary model is different than the adversary model used when proving properties about privacy or integrity protocols. It is not the model of an unknown third party intercepting or altering messages in-transit, or originating faulty messages. These topics have been well-studied both in proof systems and in implementation. We therefore assume that communication channels are free from tampering and eavesdropping. We justify this assumption by noting that implementations of decentralized, distributed systems operate under existing cryptographic protocols like TLS [45] that provide these guarantees. We also assume that adversaries cannot create new identities for themselves to launch a Sybil attack [48]. It is not the intent of this logic to prove cryptographic properties about protocols. We refer the reader to the logics mentioned above and in section 7.2 for that purpose.

We need a logic that allows us to analyze systems where we no longer assume that all participants are altruistic but instead self-interested, and can act greedily contrary to common goals. In these systems, known and static trust relationships are the exception, not the norm, and it is observed behavior that forms the basis of trust.

For instance, if two services A, B are connected, and A makes a request of B, we would like to show that, given a history of cooperation between A and B, B will honor this request; or, conversely, if there has been a lack of cooperation, B will decline it. We would also like to show that capabilities can be delegated in a way that enables composition, but that also respects restrictions imposed by the providing service.

We give a logic that unifies the logic of authentication due to Burrows, Abadi, and Needham [33], colloquially known as "BAN logic," with past- and future- time temporal logic [50, 52, 104]. We incorporate the notion of belief from BAN logic to draw a distinction between that which is knowably true and that which is believed to be true by a service. We also incorporate the notion of jurisdiction from BAN logic to allow us to deal directly with delegations of decision-making. We combine these with temporal logic to reason about the actions of services based on those beliefs and trust relationships. We show how beliefs lead to action or inaction, and how such action or inaction can then lead to further beliefs.

This logic enables us to express and prove properties of the contractually-limited capability-based access control given in chapter 5 using a reputation mech-

anism, such as is given in chapter 6, to decide access. We give the syntax and semantics of the logic, a series of example statements with their English equivalents, and then some example proofs in the logic. The proof that the logic is sound is in appendix C.

7.2 Related Work

Multi-Agent Temporal Logic [16, 17] employs Compositional Tree Logic (CTL) [37] with belief operators to reason about beliefs over traces in a similar fashion to the logic given here. This temporal logic is then used to extend results in CTL model checking by extending the notion of a finite state machine into a *multiagent finite state machine* (MAFSM). MAFSMs have been applied to cryptographic protocols for their verification, but also include the idea of participants in a cryptographic protocol believing that other parties in a protocol have sent or received certain messages. It has not yet been applied to decentralized networks and the issues of belief we have addressed, but its ability to be mapped to a model-checking algorithm is encouraging for applying our logic to model checking.

Despite its flaw regarding shared keys [96], BAN logic [33] is a popular logic for describing authentication protocols [14, 62, 79, 126]. From this logic we take the notions and syntax of belief and jurisdiction. Since BAN logic is focused on authentication, which only aims to prove that two parties in an interaction can be assured of the identity of the other, it lacks the ability to address networks where identity and trust can be uncertain, and where the goal is not to establish identity based on pre-existing trust relationships, but rather establish those trust relationships and ensure provision of service based on those relationships. The π-calculus [88] is designed to model mobile and communicating processes, and has also been used to verify protocols [2, 3] for specific cryptographic properties, and not address how trust is established in the first place.

Protocol Composition Logic [41] gives a general framework for deriving security protocols from simple components, and uses temporal logic to prove properties about security protocols such as Internet Key Exchange (IKE) and Just Fast Keying (JFK) [42]. Like BAN logic and the π-calculus, it is concerned with proving cryptographic properties of protocols based on known trust relationships, rather than access control properties of systems with dynamic and uncertain trust values.

7.3 Syntax

We begin by defining the syntax of the logic. A *signature* $\mathcal{G} = (S, O, R, I)$ includes four disjoint sets: a set of service identifiers S, a set of operation identifiers O, a set of relation identifiers R, and a set of state indices I. The syntax is given in figure 7.1.

$$
\begin{array}{rcl}
\langle service \rangle & ::= & \langle \text{element of } S \rangle \\
\langle operationid \rangle & ::= & \emptyset \mid \langle \text{element of } O \rangle \\
\langle operation \rangle & ::= & \langle service \rangle_{\langle operationid \rangle} \\
\langle opargs \rangle & ::= & \overline{\langle operation \rangle (\langle term - list \rangle)} \\
\langle capability \rangle & ::= & \overline{\langle operation \rangle \, [\langle formula \rangle]} \\
\langle capargs \rangle & ::= & \langle capability \rangle (\langle term - list \rangle) \\
\langle term \rangle & ::= & \langle service \rangle \mid \langle operation \rangle \mid \langle capability \rangle \mid \langle capargs \rangle \mid \langle opargs \rangle \\
\langle term - list \rangle & ::= & \langle term \rangle \ , \ \langle term - list \rangle \mid \langle term \rangle \mid \\
\langle action \rangle & ::= & !\langle opargs \rangle \mid \langle service \rangle \ \mathrel{|\!\sim} \ (\langle service \rangle, \langle message \rangle) \\
\langle message \rangle & ::= & \langle capability \rangle \mid \langle capargs \rangle \mid \langle opargs \rangle \mid \langle formula \rangle \\
\langle relation \rangle & ::= & \texttt{connected} \mid \langle \text{element of } R \rangle \\
\langle temporal - op \rangle & ::= & \bigcirc \mid \Diamond \mid \Box \mid \odot \mid \diamondsuit \mid \boxdot \\
\langle rel - op \rangle & ::= & = \mid < \mid > \\
\langle formula \rangle & ::= & \langle relation \rangle (\langle term - list \rangle) \mid \langle term \rangle \langle rel - op \rangle \langle term \rangle \mid \\
 & & \langle formula \rangle \vee \langle formula \rangle \mid \langle formula \rangle \wedge \langle formula \rangle \mid \\
 & & \langle formula \rangle \rightarrow \langle formula \rangle \mid \langle temporal - op \rangle \langle formula \rangle \mid \\
 & & \langle formula \rangle \ \mathbf{S}_w \ \langle formula \rangle \mid \langle formula \rangle \ \mathbf{S}_s \ \langle formula \rangle \mid \\
 & & \langle service \rangle \triangleright \langle capability \rangle \mid \langle service \rangle \ \Rrightarrow \ \langle formula \rangle \mid \\
 & & \langle service \rangle \overrightarrow{\sim} (\langle service \rangle, \langle message \rangle) \mid \neg \langle formula \rangle \mid \\
 & & \langle service \rangle \models \langle formula \rangle \mid \langle message \rangle \in K_{\langle service \rangle} \mid \\
 & & \langle action \rangle \\
\langle state \rangle & ::= & \gamma_{\langle \text{element of } I \rangle} \\
\langle statement \rangle & ::= & \langle state \rangle \models \langle formula \rangle
\end{array}
$$

Figure 7.1: Syntax of the Service Logic

7.3.1 Informal Description of Syntax

An operation identifier k represents some functionality implemented by a service, and is denoted by the service name with the operation name subscripted, like s_k.

Each service s implements a "handshake" operation represented by $s_\emptyset$, which is how a client requests an access credential.

A capability is a credential that provides access to an operation, and is represented by an operation with an attached formula, with a bar over them both: $\overline{s_k[\mathcal{C}]}$. This formula is a contract, which is a formula that must be true in order for the capability to be used. A capability paired with a set of arguments, used during the operation to evaluate the contract and make an invocation, is denoted by appending a parenthesized list of arguments to a capability: $\overline{s_k[\mathcal{C}]}(P)$

A term is a service, an operation, a capability, a capability with arguments, or an operation result.

We introduce two actions. The first, !, when applied to an operation denotes that operation has been executed. The second, $\mathrel{|\!\sim}$, used in the form $s \mathrel{|\!\sim} (u, m)$ denotes that a service s sends message m to service u. A message is either a capability, a capability paired with arguments, an operation result, or a formula.

We use the following operators from past- and future-time temporal logic:

- $\bigcirc F$: F holds in the next state
- $\Diamond F$: F holds eventually in the future
- $\Box F$: F holds always in the future (in each future state)
- $F \; \mathcal{U}_w \; G$: Either F holds forever, or when F does not hold, G holds,
- $F \; \mathcal{U}_s \; G$: F holds until G holds, and eventually G holds
- $\odot F$: F held in the immediately preceding state
- $\diamondsuit F$: F held at some state in the past
- $\boxdot F$: F held at all states in the past
- $F \; \mathcal{S}_w \; G$: Either F always held, or when F no longer held G has held since,
- $F \; \mathcal{S}_s \; G$: F held until G held, and eventually G held

A formula is then:

- A relation on a list of terms,

- The result of an operation s_k executed over parameters P $S_k(P)$,
- The negation of a formula $\neg F$,
- The disjunction $F \vee G$ or the conjunction $F \wedge G$ of two formulas,
- The implication of formulas $F \rightarrow G$,
- The temporal quantification of a formula,
- The belief of a service S in a formula F, $S \models F$,
- The knowledge of a service S of a formula F, $F \in K_S$,
- The belief that one service C is entitled to a capability X, $C \triangleright X$,
- The belief that a service S has jurisdiction over a formula F, $S \Rightarrow F$, or
- The intent of a service C to send a message M to another service S, $C \overrightarrow{\sim}(S, M)$.

A statement is either that the selected action in a state γ_i is an action A, or that in γ_i a formula F holds.

7.4 Semantics

The semantics of this logic are defined on traces of execution in a Kripke structure. Each element of a trace $x = (\gamma_0, \gamma_1, \gamma_2, \ldots)$ is a global state. We use the syntax $\gamma_i \models F$ to state that a formula F holds in state γ_i. Each γ_i contains:

- The local state of each individual service s, including
  - The *knowledge set* K_s, containing capabilities and data known by s,
  - The *belief set* B_s, containing what formulas s believes to be true,
- The *truth set* $\mathcal{T}$, containing predicates that are globally true in that state,
- One or more *enabled actions* that are ready to be executed in that state,
- The *selected action*, chosen nondeterministically, that causes the transition to the next state.

We assume the asynchronous interleaved execution model, which states that no two actions actually occur simultaneously, but that any of the enabled actions in a given state may be selected for the next transition. This model allows tractable reasoning about systems by dealing with only one action at a time, recognizing that each individual service is a sequential machine and therefore the simultaneous execution of these services can be accurately modeled by an arbitrary interleaving of their sequential actions.

There are two possible actions that cause a state transition:

- For any operation s_o, where $s \in S$ and $o \in O$, the expression $\gamma_i \models !s_o(P)$ indicates that the selected action of state γ_i is the execution of the operation o by service s with parameters P.
- For any two services s, c and any formula m, the expression $\gamma_i \models s\,|\!\sim(c, m)$ indicates that the selected action of state γ_i is the sending of the message m from service s to service c.

For a service s and a formula f, the formula $\gamma_i \models s \mid\!\equiv f$ indicates that $\gamma_i \models f \in B_s$.

In these two situations, we refer to a particular state γ_i. We may refer to more states by combining these formulas with temporal operators. We define the semantics for the following formulas [50, 52]:

- $\gamma_i \models \bigcirc F \Leftrightarrow \gamma_{i+1} \models F$
- $\gamma_i \models \Diamond F \Leftrightarrow (\exists j \geq i)\gamma_j \models F$
- $\gamma_i \models \Box F \Leftrightarrow (\forall j \geq i)\gamma_j \models F$
- $\gamma_i \models F\ \mathcal{U}_s\ G \Leftrightarrow (\exists j \geq i)(\gamma_j \models G) \wedge (\forall k, i \leq k < j)(\gamma_k \models F)$
- $\gamma_i \models F\ \mathcal{U}_w\ G \Leftrightarrow \gamma_i \models (F\ \mathcal{U}_s\ G \vee \Box F)$
- $\gamma_i \models \odot F \Leftrightarrow \gamma_{i-1} \models F$
- $\gamma_i \models \Diamond\!\!\!\!-\ F \Leftrightarrow (\exists j \leq i)\gamma_j \models F$
- $\gamma_i \models \boxdot F \Leftrightarrow (\forall j \leq i)\gamma_j \models F$

- $\gamma_i \models F\ \mathcal{S}_s\ G \Leftrightarrow (\exists j \leq i)(\gamma_j \models G) \wedge (\forall k, j < k \leq i)(\gamma_k \models F)$
- $\gamma_i \models F\ \mathcal{S}_w\ G \Leftrightarrow \gamma_i \models (F\ \mathcal{S}_s\ G \vee \boxdot F)$

In addition to application-specific relations in the set R, we assume the standard relations $=, <, >$ and introduce the following:

- $s \Rrightarrow f$: s has "jurisdiction" over f: if s believes f, then any t that believes $s \Rrightarrow f$ will also believe f upon learning that s believes f. This can only appear in belief sets of an individual service, though not necessarily that of s.
- $c \triangleright x$: For a service c and a capability x, this indicates that c is trusted to receive capability x. This only appears in the belief set of an individual service, though not necessarily that of c.
- $c \overrightarrow{\sim} (s, m)$: For service c, s and a message m, this indicates that c intends to send m to s. This only appears in the belief set of the service c.
- `connected(s, c)`: For two services $s, c \in S$, there exists a reliable network path between s and c. Since we assume these paths are bi-directional, `connected` is a symmetric relation. This only appears in the truth set of a state.

The *handshake axiom* 7.1 encodes the assumption that in any state, any client can create a capability to the handshake operation on any service:

$$\gamma_i \models \overline{s_\emptyset[\top]} \in K_c \tag{7.1}$$

The *capability axiom* 7.2 encodes the assumption that in any state, a service can create a capability to any operation it provides with any contract:

$$\gamma_i \models \overline{s_k[C]} \in K_s \tag{7.2}$$

We introduce the following rules of inference. First, we give the *connectedness* rule 7.3. If s sends a message m to c, and there is a network path between s and c (they are *connected*), then eventually c believes that s sent m to it. Again, recall that our adversary model does not allow interfering or modifying messages in transit.

$$\frac{\begin{array}{l}\gamma_i \models s \mathrel{|\!\sim} (c, m) \\ \gamma_i \models \texttt{connected}(s, c)\end{array}}{\gamma_{i+1} \models \Diamond c \mid\!\equiv s \mathrel{|\!\sim} (c, m)} \tag{7.3}$$

The *says-belief* rule 7.4 states if a service s tells a service c that it believes a formula f, c believes that s believes f.

$$\frac{\gamma_i \models c \mid\!\equiv s \mathrel{|\!\sim} (c, s \mid\!\equiv f)}{\gamma_i \models c \mid\!\equiv s \mid\!\equiv f} \tag{7.4}$$

To relate past-time and future-time operators when specifying message transmission, we introduce the *change of frame* rule 7.5 to relate two global states. It is not required that $s = v$, although frequently this is the case. m is any message.

$$\frac{\gamma_i \models \Diamond s \mid\!\equiv c \mathrel{|\!\sim} (v, m)}{\begin{array}{l}(\exists j) j > i \\ \gamma_j \models s \mid\!\equiv \Diamond c \mathrel{|\!\sim} (v, m)\end{array}} \tag{7.5}$$

Next, we give the *sending* rule 7.6. If c knows a message m, and it intends to send m to a service s, in a later state c sends m to s.

$$\frac{\begin{array}{l}\gamma_i \models m \in K_c \\ \gamma_i \models c \overrightarrow{\mathrel{|\!\sim}} (s, m)\end{array}}{\begin{array}{l}(\exists j) j > i \\ \gamma_j \models c \mathrel{|\!\sim} (s, m)\end{array}} \tag{7.6}$$

Similar to the sending rule is the *send-belief* rule 7.7. If c believes formula f, and c intends to send to s the fact that it believes f, in a later state it sends that belief to s.

$$\frac{\begin{array}{l}\gamma_i \models c \mid\!\equiv f \\ \gamma_i \models c \overrightarrow{\mathrel{|\!\sim}} (s, c \mid\!\equiv f)\end{array}}{\begin{array}{l}(\exists j) j > i \\ \gamma_j \models c \mathrel{|\!\sim} (s, c \mid\!\equiv f)\end{array}} \tag{7.7}$$

Next, we give the *handshake* rule 7.8. If s believes c is trusted to receive a capability $\overline{s_i[\mathcal{C}]}$ with some contract $\mathcal{C}$, and c invokes the handshake operation on s for access

to operation s_i, in the next state, s will send that capability to c.

$$\frac{\begin{array}{l}\gamma_i \models s \mid\!\equiv c \triangleright \overline{s_k[\mathcal{C}]} \\ \gamma_i \models s \mid\!\equiv \diamond c \mid\!\sim (s, \overline{s_\emptyset[\top]}(s_k))\end{array}}{\begin{array}{l}(\exists j) j > i \\ \gamma_j \models s \mid\!\sim (c, \overline{s_k[\mathcal{C}]})\end{array}} \tag{7.8}$$

Next, we give the *delegation* rule 7.9. This simply states that if c believes it received a capability for s_k from s, that capability is part of its knowledge set.

$$\frac{\gamma_i \models c \mid\!\equiv \diamond s \mid\!\sim (c, \overline{s_k[\mathcal{C}]})}{\gamma_i \models \overline{s_k[\mathcal{C}]} \in K_c} \tag{7.9}$$

Next, we give the *invocation* rule 7.10. This states that if s receives an invocation request with a capability, and it believes the contract is satisfied, then it performs the operation in some future state.

$$\frac{\begin{array}{l}\gamma_i \models s \mid\!\equiv \diamond c \mid\!\sim (s, \overline{s_k[\mathcal{C}]}(P)) \\ \gamma_i \models s \mid\!\equiv \mathcal{C}\end{array}}{\begin{array}{l}(\exists j) j > i \\ \gamma_j \models !s_k(P)\end{array}} \tag{7.10}$$

Related to the invocation rule is the *result* rule 7.11. This states that if s executes operation s_k with parameters P, then in the immediately following state it knows the result of that operation $s_k(P)$.

$$\frac{\gamma_i \models !s_k(P)}{\gamma_{i+1} \models s_k(P) \in K_s} \tag{7.11}$$

The trivial contact, represented by $\top$, is always satisfied. This is represented by the *trivial contract* axiom:

$$\gamma_i \models s \mid\!\equiv \top \tag{7.12}$$

Finally, we give the *jurisdiction* rule. If c believes that s has jurisdiction over a formula f, and c believes that s believes f, then in the following state, c also believes f.

$$\frac{\gamma_i \models c \models s \Rightarrow f \quad \gamma_i \models c \models s \models f}{\gamma_i \models c \models f} \tag{7.13}$$

7.5 Example Formulas

To help understand the logic, we provide here a number of example formulas written in the logic.

In these examples, A and B will be two services (remember, clients are services) interacting with one another. These are not all necessarily formulas we expect to be true, as some are a bit too simple to actually happen in a real system, but are offered as examples.

1. A is cooperative.

$$\texttt{cooperative}(A)$$

2. B believes A is cooperative.

$$B \models \texttt{cooperative}(A)$$

3. A will provide B with capability $\overline{A_k[\mathcal{C}]}$. In other words, A believes that B is entitled to capability $\overline{A_k[\mathcal{C}]}$.

$$A \models B \triangleright \overline{A_k[\mathcal{C}]}$$

4. B sends a request to A for operation A_k with some parameters P.

$$B \mid\sim (A, \overline{A_k[\mathcal{C}]}(P))$$

5. If B previously requested A_k from A, and A_k was performed as a result, then B believes A is cooperative.

$$\diamond(B \mid\sim (A, \overline{A_k[\mathcal{C}]}(P)) \wedge \diamond !A_k(P)) \rightarrow \diamond B \models \texttt{cooperative}(A)$$

6. If it is always the case that when B requests A_k from A, and A performs A_k, then eventually B will believe A is cooperative.

$$\Box\left((B \mid\sim (A, \overline{A_k[\mathcal{C}]}(P)) \wedge \Diamond !A_k(P)) \rightarrow \Diamond B \models \texttt{cooperative}(A)\right)$$

7. If A believes B is cooperative, and A believes A_k should be offered to cooperative peers, then A will provide $\overline{A_k[\mathcal{C}]}$ to B.

$$A \models \texttt{cooperative}(B) \wedge A \models \texttt{provideWhenCooperative}(A_k) \rightarrow A \models B \triangleright \overline{A_k[\mathcal{C}]}$$

8. If A believes B is cooperative, and B requests A_k from A with parameters P, then eventually B believes A is cooperative.

$$A \models \texttt{cooperative}(B) \wedge B \mid\sim (A, \overline{A_k[\mathcal{C}]}(P)) \rightarrow \Diamond B \models \texttt{cooperative}(A)$$

9. If B requests A_k from A with parameters P, and A is malicious, A_k will not be executed.

$$B \mid\sim (A, \overline{A_k[\mathcal{C}]}(P)) \wedge \texttt{malicious}(A) \rightarrow \neg\Diamond !A_k(P)$$

So far we have left the precise meaning of the predicate `cooperative` as unspecified, but whether or not a service is considered cooperative is crucial to decisions involving its trust. We therefore provide additional predicates concerning the determination of cooperation.

10. A has serviced k of n service requests. (We assume the others were either ignored, or lost due to benign failure.)

$$\texttt{hasServiced}(A, k, n)$$

11. B believes A has serviced k of n requests (presumably, ones that B made).

$$B \models \texttt{hasServiced}(A, k, n)$$

12. B believes A has serviced at least half of its service requests.

$$B \models \texttt{hasServiced}(A, k, n) \wedge k \geq n/2$$

13. If B believes A has serviced at least half of its requests, then B believes A is cooperative.

$$B \models \texttt{hasServiced}(A, k, n) \wedge k \geq n/2 \rightarrow B \models \texttt{cooperative}(A)$$

14. If A is altruistic, and A believes B has requested service A_k with parameters P, A_k will be executed.

$$\texttt{altruistic}(A) \wedge A \models B \mathrel{|\!\sim} (A, \overline{A_k[\mathcal{C}]}(P)) \rightarrow \Diamond !A_k(P)$$

15. If A believes B has requested service A_k some time in the past with parameters P, and A believes B is cooperative, then A_k will be executed in the following state.

$$A \models \Diamond B \mathrel{|\!\sim} (A, \overline{A_k[\mathcal{C}]}(P)) \wedge A \models \texttt{cooperative}(B) \rightarrow \Diamond !A_k(P)$$

We can now reason about the number of successful requests for a particular service, as opposed to those provided by a service.

16. B successfully received service on k of its n requests to A.

$$B \models \texttt{receivedService}(A, k, n)$$

17. B successfully received service on at least half of its n requests to A.

$$B \models \texttt{receivedService}(A, k, n) \wedge k \geq n/2$$

Let us now suppose A, E are both sources of data flows. Suppose A is in Asia and E is in Europe. Let C be a client consuming both flows. Let us say that a data flow is *sweet* when the flow provides useful data, and *sour* otherwise.

18. A sweet Asian flow that has always been sweet means that C believes the Asian service is cooperative.

$$C \models \boxdot \texttt{sweet}(A) \rightarrow C \models \texttt{cooperative}(A)$$

19. If the European flow has ever been sour, then C believes the European service is not cooperative.

$$C \models \diamond \texttt{sour}(E) \rightarrow C \models \neg \texttt{cooperative}(E)$$

7.6 Example Proofs

We give proofs of the following theorems:

- Suppose A and B are services. A will provide service to B if the B has proven cooperative in the past.
- Suppose A believes B should have access to operation A_k, and B believes C should as well. Through a series of delegations, C can invoke A_k.
- If a service defers its decisions to an authorization server, and the authorization server believes a client should be authorized, then that client can invoke that operation.

Theorem 7.6.1 *Given that A and B are connected, if A requested operation B_k from B, and A believes B_k was performed, then A believes B is cooperative.*

For this example, we must provide a criterion for when a service is deemed cooperative. We introduce two predicates: $\texttt{cooperativeCount}(s, n)$ that tracks the number of interactions in which s has been cooperative, and $\texttt{uncooperativeCount}(s, n)$ which tracks the number where it has not been. A believes `cooperative(B)` if and only if the `cooperativeCount` is greater than the `uncooperativeCount`.

We assume A has never asked B for any other service, so both counts begin at zero. The proof then is as follows. Note that $i < j$.

1.	$\gamma_i \models \texttt{connected}(A, B)$	*given*
2.	$\gamma_i \models \diamond A \mid\!\sim (B, \overline{B_k[\mathcal{C}]}(P))$	*given*
3.	$\gamma_i \models B \mid\!\sim (A, F)$	*given*
4.	$\gamma_{i+1} \models \diamond A \equiv B \mid\!\sim (A, F)$	*connectedness rule*
5.	$\gamma_j \models A \equiv \diamond B \mid\!\sim (A, F)$	*change of frame rule*
6.	$\gamma_j \models A \equiv F = B_k(P)$	*given*
7.	$\gamma_j \models A \equiv \texttt{cooperativeAt}(\texttt{B}, \texttt{j})$	*def'n of* `cooperativeAt`
8.	$\gamma_j \models A \equiv \texttt{cooperativeCount}(\texttt{B}, \texttt{1})$	*def'n of* `cooperativeCount`
9.	$\gamma_j \models A \equiv \texttt{uncooperativecount}(\texttt{B}, \texttt{0})$	*given*
10.	$\gamma_j \models A \equiv \texttt{cooperative}(\texttt{B})$	*def'n of* `cooperative`

□

Theorem 7.6.2 *Given that A, B, C are connected, if A believes B should have unrestricted access to operation A_k, and B believes C should have access to operation A_k, through a series of delegations and handshakes, C can invoke service A_k.*

Note that $i < j < m < n < p < q < r < s < t < u$.

1.	$\gamma_i \models \texttt{connected}(A, B)$	*given*
2.	$\gamma_i \models A \equiv B \triangleright \overline{A_k[\top]}$	*given*
3.	$\gamma_i \models \overline{A_\emptyset[\top]} \in K_B$	*handshake axiom*
4.	$\gamma_i \models B \overrightarrow{\sim} (A, \overline{A_\emptyset[\top]}(A_k))$	*given*
5.	$\gamma_j \models B \mid\!\sim (A, \overline{A_\emptyset[\top]}(A_k))$	*sending rule*
6.	$\gamma_{j+1} \models \diamond A \equiv B \mid\!\sim (A, \overline{A_\emptyset[\top]}(A_k))$	*connectedness rule*
7.	$\gamma_m \models A \equiv \diamond B \mid\!\sim (A, \overline{A_\emptyset[\top]}(A_k))$	*change of frame rule*
8.	$\gamma_n \models A \mid\!\sim (B, \overline{A_k[\top]})$	*handshake rule*
9.	$\gamma_{n+1} \models \diamond B \equiv A \mid\!\sim (B, \overline{A_k[\top]})$	*connectedness rule*
10.	$\gamma_p \models B \equiv \diamond A \mid\!\sim (B, \overline{A_k[\top]})$	*change of frame rule*
11.	$\gamma_p \models \overline{A_k[\top]} \in K_B$	*delegation rule*
12.	$\gamma_p \models B \overrightarrow{\sim} (C, \overline{A_k[\top]})$	*given*
13.	$\gamma_q \models B \mid\!\sim (C, \overline{A_k[\top]})$	*sending rule*
14.	$\gamma_q \models \texttt{connected}(B, C)$	*given*
15.	$\gamma_{q+1} \models \diamond C \equiv B \mid\!\sim (C, \overline{A_k[\top]})$	*connectedness rule*
16.	$\gamma_r \models C \equiv \diamond B \mid\!\sim (C, \overline{A_k[\top]})$	*change of frame rule*

17. $\gamma_r \models \overline{A_k[\top]} \in K_C$ *delegation rule*
18. $\gamma_r \models C \overrightarrow{\sim}(A, \overline{A_k[\top]}(P))$ *given*
19. $\gamma_s \models C \sim(A, \overline{A_k[\top]}(P))$ *sending rule*
20. $\gamma_s \models \texttt{connected}(C, A)$ *given*
21. $\gamma_{s+1} \models \Diamond A \equiv C \sim(A, \overline{A_k[\top]}(P))$ *connectedness rule*
22. $\gamma_t \models A \equiv \diamond C \sim(A, \overline{A_k[\top]}(P))$ *change of frame rule*
23. $\gamma_t \models A \equiv \top$ *trivial contract axiom*
24. $\gamma_u \models !A_k(P)$ *invocation rule*

□

Theorem 7.6.3 *Let S be a service that provides an operation S_k, and let A be an authorization server to which S defers authorization decisions about S_k. We show that if all three services are connected, if A believes client C should have access to S_k with some contract $\mathcal{C}$, and $\mathcal{C}$ is satisfied, C can invoke S_k.*

We introduce relation $\texttt{isAuthorizedQuery}(\texttt{C}, \texttt{S}_\texttt{k})$, which is the query S makes to A. We assume S knows how to make these queries. Note that $i < j < m < n < p < q < r < s < t < u < v$.

1. $\gamma_i \models \overline{S_\emptyset[\top]} \in K_C$ *handshake axiom*
2. $\gamma_i \models C \overrightarrow{\sim}(S, \overline{S_\emptyset[\top]}(S_k))$ *given*
3. $\gamma_j \models C \sim(S, \overline{S_\emptyset[\top]}(S_k))$ *sending rule*
4. $\gamma_j \models \texttt{connected}(C, S)$ *given*
5. $\gamma_{j+1} \models \Diamond S \equiv C \sim(S, \overline{S_\emptyset[\top]}(S_k))$ *connectedness rule*
6. $\gamma_m \models S \equiv \diamond C \sim(S, \overline{S_\emptyset[\top]}(S_k))$ *change of frame rule*
7. $\gamma_m \models \texttt{isAuthorizedQuery}(\texttt{C}, \texttt{S}_\texttt{k}) \in K_s$ *given*
8. $\gamma_m \models S \overrightarrow{\sim}(A, \texttt{isAuthorizedQuery}(\texttt{C}, \texttt{S}_\texttt{k}))$ *given behavior*
9. $\gamma_n \models S \sim(A, \texttt{isAuthorizedQuery}(\texttt{C}, \texttt{S}_\texttt{k}))$ *sending rule*
10. $\gamma_n \models \texttt{connected}(\texttt{S}, \texttt{A})$ *given*
11. $\gamma_{n+1} \models \Diamond A \equiv S \sim(A, \texttt{isAuthorizedQuery}(\texttt{C}, \texttt{S}_\texttt{k}))$ *connectedness rule*
12. $\gamma_p \models A \equiv \diamond S \sim(A, \texttt{isAuthorizedQuery}(\texttt{C}, \texttt{S}_\texttt{k}))$ *change of frame rule*
13. $\gamma_p \models A \equiv C \triangleright \overline{S_k[\mathcal{C}]}$ *given*
14. $\gamma_p \models A \overrightarrow{\sim}(S, A \equiv C \triangleright \overline{S_k[\mathcal{C}]})$ *given behavior*
15. $\gamma_q \models A \sim(S, A \equiv C \triangleright \overline{S_k[\mathcal{C}]})$ *send − belief rule*
16. $\gamma_{q+1} \models \Diamond S \equiv A \sim(S, A \equiv C \triangleright \overline{S_k[\mathcal{C}]})$ *connectedness rule*

17. $\gamma_r \models S \models \diamond A \mid\sim (S, A \models C \triangleright \overline{S_k[\mathcal{C}]})$ *change of frame rule*
18. $\gamma_r \models S \models A \Rightarrow C \triangleright \overline{S_k[\mathcal{C}]}$ *given*
19. $\gamma_r \models S \models A \models C \triangleright \overline{S_k[\mathcal{C}]}$ *says − belief rule*
20. $\gamma_r \models S \models C \triangleright \overline{S_k[\mathcal{C}]}$ *jurisdiction rule*
21. $\gamma_s \models S \mid\sim (C, \overline{S_k[\mathcal{C}]})$ *handshake rule*
22. $\gamma_s \models \texttt{connected}(\texttt{S}, \texttt{C})$ *given*
23. $\gamma_{s+1} \models \diamond C \models S \mid\sim (C, \overline{S_k[\mathcal{C}]})$ *connectedness rule*
24. $\gamma_t \models C \models \diamond S \mid\sim (C, \overline{S_k[\mathcal{C}]})$ *change of frame rule*
25. $\gamma_t \models \overline{S_k[\mathcal{C}]} \in K_C$ *delegation rule*
26. $\gamma_t \models C \mid\sim (S, \overline{S_k[\mathcal{C}]}(P))$ *given behavior*
27. $\gamma_{t+1} \models \diamond S \models C \mid\sim (S, \overline{S_k[\mathcal{C}]}(P))$ *connectedness rule*
28. $\gamma_u \models S \models \diamond C \mid\sim (S, \overline{S_k[\mathcal{C}]}(P))$ *change of frame rule*
29. $\gamma_u \models S \models \mathcal{C}$ *given*
30. $\gamma_v \models !S_k(P)$ *invocation rule*

□

7.7 Conclusions

We have given a logic that unifies temporal logic with key ideas from BAN logic. This logic allows formulation of properties of interest in decentralized systems of services. We have given the syntax and semantics of the logic, and provided example statements with their English equivalents, and then proved three theorems. We prove the logic is sound in appendix C.

In the next chapter, we conclude this dissertation with a summation of all the work contributed, and list the many possible directions of future research suggested.

Chapter 8

Conclusions and Future Work

Distributed systems with decentralized control are playing an increasing role in cooperative systems of all kinds, from loose confederations of anonymous participants on the Internet, to cross-organization grid collaborations. As these systems grow more decentralized, they require novel approaches for access control to services they provide. Mechanisms that cope well with both the changing and unbounded size of these networks, as well as the uncertainty of trust inherent in any relationship, are needed to support secure interactions in these systems. Access control is no longer the gatekeeper that interacts with clients only to check credentials at the point of entry and then interacts with them no more, but one of the many players interacting inside the system. Instead of an isolated subsystem, access control is now amongst the requirements and correctness conditions that are woven into the core of a system.

This dissertation has examined the fundamental issues of access control in these systems by constructing a taxonomy, identified approaches not yet explored in that space, and given a method of comparing tradeoffs for a number of design decisions. Based on CoorSet, a system for computation in decentralized systems, we introduced contractually-limited capabilities, a method for accomplishing access control that functions well under the requirements of decentralized systems. We then augmented this mechanism with a reputation tracking scheme that quantifies uncertainty in its ratings to accommodate the incomplete state intrinsic to these systems. Finally, we gave a means of formally verifying properties of these systems in a mathematical logic.

8.1 Future Research Directions

We have covered a wide range of topics in this dissertation. Here we present future directions for research in each of the major topics explored.

8.1.1 Taxonomy of Access Control

There may exist further properties of implementations that can define additional axes. There may exist additional points representing additional design choices on the axes discovered so far. In particular, the Trust Management axis may have mechanisms for implementation of adaptation which lead to different degrees of fidelity of enforcement. Currently, any trust management approach that is at all adaptive is classified under that point. However there are multiple kinds of adaptive solutions: fully automated ones, and those which use some input from an external source such as a user or administrator. Therefore, the Adaptive point may be decomposed into additional points. We will continue to look for these as future work.

In an ideal taxonomy the axes of the lattice would be orthogonal from each other, but we can see that our current axes are not. In particular, we have seen an interplay between the Decision Mode and the Enforcement axis: in the systems with Centralized decision mode, there exists the potential for a network failure where the central server is inaccessible, and so a service will default to "fail-closed" rather than possibly improperly service a request that limits the implementation to total prevention on the Enforcement axis. We continue to look for orthogonal decompositions of this space, in which some of these non-orthogonal axes may instead be better expressed as combinations of more fundamental, orthogonal ones.

Although a taxonomy such as this is interesting as a way of formalizing properties that so far have been treated informally, its possible use as a means of comparing implementations of a particular access control scheme. We may wish to compare properties such as vulnerability to a given form of attack, or motivating exploration of alternative implementation choices for it. Specifically, we are looking into creating a set of metrics that give ordered sub-lattices of this taxonomy with respect to how well an implementation guards a system against particular kinds of threats, like denial of service, unintended provision of service, unintended information flow, or collusion amongst a group of adversaries. Enforcement may itself be

such a metric, rather than an axis, that is measured over a number of more basic properties yet to be discovered.

We have only seen one access control implementation so far attempt to take advantage of decentralized state distribution: dRBAC. Solutions requiring no state storage on the side of the server have shown resilience to Denial of Service attacks. These solutions employ cryptography to store state in an immutable way on the client, just as the certificate chains in dRBAC, the one system we have found at this state distribution point. Conversely, such increased distribution could lead to unintended information flow, where access rights or trust relationship information is exposed. In systems where colluding groups of adversaries is an issue, adaptive trust management and more global decision modes can lead to quicker detection of such groups. We will explore formalizing these metrics in terms of our axes.

8.1.2 Contractually-Limited Capabilities and CapCoorSet

Contracts are currently a set of conditions and constraints placed on the holder of the capability, which the service agrees to honor if those conditions are satisfied. We are investigating extending contracts to conditions on both sides (for example, *service level agreements*, or SLA's), so that not only are requirements for use imposed on the invoker, but the service also has terms describing its behavior and service that must be satisfied. We are also exploring how to incorporate additional supporting evidence for authentication both as part of the handshake for negotiating new capabilities, and other kinds of supporting evidence to prove satisfaction of contracts. This will lead to more elaborate negotiations, instead of the two-step handshake we currently use.

8.1.3 Automated Trust Decisions

The work on reputation tracking suggests several directions for future work, including using uncertainty in reputation computations. We will consider networks that support more than one generic operation, and introduce more than one factor δ to represent the change in reputation resulting from differing levels of cooperation. For instance, cooperation or lack thereof in routing a single packet may warrant a relatively small change in reputation, while cooperation in performing an intensive computation may warrant a large change in reputation.

Currently, neighborhood query results are used regardless of how much they differ from the local rating. This rating may deviate greatly from the local rating, which can indicate that the neighborhood rating should be further discounted or ignored entirely. This can be an issue in networks with particularly high numbers of uncooperative services. In the future we will investigate the effect of such a test on the efficacy of neighborhood queries on overall performance.

Jøsang et. al. [69] observe that trust to perform an action is not the same as trust to recommend another to perform an action. In fact, there are many different kinds of trust, and different actions may not have sufficient *semantic constraints* [70] to be relatable under a single measure of trust. We will explore maintaining multiple opinions rather than one, not only for trust to perform various kinds of actions, but trust to provide recommendations as well.

Simulations fixed the cooperative population of the network at 75%. Although cooperative services do not innately know who is and who is not cooperative, this still represents a trusted core that may not be reasonable for decentralized systems in general. Simulations with other population distributions are required to justify the generality of this approach.

We would also like to study more elaborate adversary models. Not yet studied is the phenomenon of *traitors*, services which act cooperatively to build up a positive reputation and then turn suddenly uncooperative to attempt to profit from it, but we do not address this case here. Traitors may oscillate between behaviors, cooperating when the level of service received is unacceptably low, and returning to greediness afterwards. It is of interest to know how often an uncooperative participant must remain cooperative to receive an acceptable level of service. An effective cooperation mechanism would require the time spent cooperatively to be high.

Preliminary experiments suggest this behavior causes the time to convergence to be longer, and the percentage of successful requests made after the service becomes uncooperative is higher as it takes longer for an entirely positive opinion to degrade sufficiently, but if the service remains uncooperative, it is detected eventually. Tracking the variance of opinions may help further in detecting traitors, as their opinions should vary more than the opinions of cooperative services. There are several options for tracking variances. The variance in the local trust metric or in the confidence rating are two other possibilities. This could then result in the uncertainty weight for these services being lowered, to represent the pessimistic

outlook for those services' cooperation.

Although the comparison against the two other methods presented here is encouraging, comparison against other published methods, such as the ones highlighted in section 6.2, would be helpful. Future work therefore will include adapting such methods, such as CONFIDANT [31], into the simulation architecture and gathering results on their performance for evaluation against our method.

Finally, our simulation architecture was intentionally simple so we could examine just the effects of the parameters of interest under the assumption of perfect connectivity. As this work matures and provides greater insight into what techniques are more effective, we will then use more realistic simulations of ad-hoc networks in a wireless scenario to further substantiate the utility of these techniques. This will include addressing specific network and adversary topologies, to examine the effects of situations such as adversaries concentrated in one location versus distributed throughout the network. Additionally, although we have shown that neighborhood queries help throttle uncooperative services, we have not examined the cost of queries on the network, particularly in the beginning of its operation where everyone is unknown. Using uncertainty to guide when network queries are used may help minimize the load on the network from such queries. Limiting the scope of such queries may also help reduce this load while maintaining the effectiveness of those queries on performance. A more realistic simulator will allow this to be studied.

8.1.4 Verification of Service-Oriented Systems

The logic allows us to state and prove useful properties about service-oriented systems. Further examples of proofs will help prove its usefulness, as well as suggest further additions to the logic. Proofs are currently done by hand, however, in a time-consuming fashion; even the relatively simple proofs in chapter 7 take up most of a page. Temporal logics have been mapped into model checkers [37]. Future-time linear temporal logic, which we use here, has been implemented in the model checker SPIN [65]. Past- and future-time temporal logic have been implemented in the Java PathExplorer [63] for runtime monitoring. Given these developments, we would like to explore extending an existing model checker to support our logic, or build a runtime monitoring system to attach to CapCoorSet or similar system that uses statements expressed in the logic.

Appendix A

CoorSet Interface Description Language

The CoorSet Interface Description Language provides for a quick configuration of the initial system state. It specifies the interfaces for components in the system, and by way of coordinating selectors with profiles, composes them together. It then allows all the components to be started on the network, with as many instantiations of each component as is desired. This allows for functionality to be transparently replicated for either load-balancing or fault-tolerance through redundancy. Each replica starts off with the same initial properties so that they may all be addressed as a group, but each is also given a unique identifier to allow it to be addressed individually.

The CoorSet compiler generates Java classes for each of the components, as well as a "starter" class to effect the instantiation of the components on the network. These components are designed to handle the interface side of the component, and link with programmer-provided code which supplies the functionality.

A.1 Building and Running Applications

A.1.1 Running the Compiler

The `CoorSet.ASL` class provides the compilation facility. You give it a configuration file in the CoorSet format, and it will generate multiple Java source files. Execute

it with:

```
java CoorSet.ASL ConfigurationFile.asl
```

Replace `ConfigurationFile.asl` with your filename. This will generate a `.java` file for each `target` entry in the configuration file, after which you can compile them all with `javac`.

A.1.2 Starting Your Application

You have two options for starting your application. For debugging purposes, you may start each component individually from different command lines. If you wish to do this, you must provide as the first parameter to each component a *unique* identifier. For example, when using the Component Starting Component, it assigns numerical addresses in order. The identifier can be any string (unless your application in some way depends on what the value is), and is specified as the first argument. For example, if you had a component called `MyApplication.MyComponent`, you would start it with a command line such as:

```
java MyApplication.MyComponent 0
```

This would start one instance of this component with the unique identifier `0`.

To use the automated script and the CSC, the compiler will provide a main program for you to run from the command line, which will submit requests to running CSCs in your multicast domain. If your main program was called `MyApplication.MyMainProgram`, you would simply run:

```
java MyApplication.MyMainProgram
```

Please make sure you run this program from the working directory where the `.class` files of the generated Java classes are located, so that the system can load them in and send them to the CSC.

A.1.3 Network Parameters

The first section of any CoorSet configuration file specifies global network parameters that are common to all components. Currently, they all are required and must appear in this order. The first series, given under the heading **Light-weight Reliable Multicast** is for use when components are launched either by hand or using the Component Starting Component, and communication takes place using the Light-weight Reliable Multicast Protocol [84] over IP Multicast, such as over a local area network. The second series, given under the heading **United Devices Grid MP** is for use when components are launched over the United Devices grid infrastructure, and communication takes place over uni-cast with a Directory Server. **Only use the directives from exactly one of the two sections.**

Light-weight Reliable Multicast

- `package` <Java Identifier>

  This directive specifies the package in which all generated interface classes will be placed, as well as in which class all programmer-provided functionality classes already exist.

- `target` <Java Identifier>

  The target is the name of the "starter" class that is responsible for starting all the initial components in the system. It will be placed in the above mentioned package, and will be generated with an appropriate "main" class to allow its execution from the command line.

- `group ''`<Multicast address>`''`

  All communication occurs over a reliable multicast. This directive specifies the multicast group to use. The address must be surrounded by quote marks.

- `port` <integer>

  In addition to a particular multicast group, all communication happens over a single port. This directive specifies that port.

- `ttl` <integer>

For any multicast, a Time-To-Live (TTL) counter is attached to each datagram, to limit its spread across the network. This parameter specifies that value. Be sure that the value is large enough to cover the entire radius of your multicasting domain from all points, remembering that the Internet measures these hops in terms of actual routers along the path, not just the hosts in the multicast network. This value will have to be large if there are any long-distance tunnels in use.

- `cscgroup` ‘‘<Multicast address>’’
- `cscport` <integer>
- `cscttl` <integer>

  These are the communication parameters for the Component Starting Component. The meanings of each parameter is the same as those above, but the CSC should operate on a different group and/or port than the main application. These will also be the communication parameters to use when you start CSC's in the network; see section A.3.2. The `cscgroup` parameter must have the multicast group in quote marks.

- `csckeys` ‘‘<File pathname>’’

  This is the file that will contain the private key used to sign the requests through the Component Starting Component. This pathname should be relevant on the host where the main program will be launched. The pathname must be contained in quote marks.

United Devices Grid MP

Because the United Devices grid deploys components on a variety of hosts, which rarely if ever share a multicast domain, we must instead “fake” multicast through the use of a Directory Server. The Directory Server maintains a list of the profiles of all components in operation, and provides a list of endpoints of hosts which map a particular selector for the purposes of broadcast. Individual components then directly connect to any or all of the matching components, and send their message. When this mode is selected, the underlying Associative Interface code automatically

registers itself with the Directory Server when started, and registers updates to the profile whenever attributes are added, removed, or changed.

The Directory Server must be in a statically-defined place. See section A.4 for information on starting the Directory Server.

- `package` <Java Identifier>

  This directive specifies the package in which all generated interface classes will be placed, as well as in which class all programmer-provided functionality classes already exist.

- `target` <Java Identifier>

  The target is the name of the "starter" class that is responsible for starting all the initial components in the system. It will be placed in the above mentioned package, and will be generated with an appropriate "main" class to allow its execution from the command line.

- `dshost` <host name>

  This specifies the name of the host on which the Directory Server is running.

- `dsport` <integer>

  This specifies the port the Directory Server is listening on.

- `sport` <integer>

  This specifies the port that each *component* will listen on for inbound messages. This can be the same as `dsport`, provided the host the Directory Server runs on is not in the United Devices grid. It must be an unused port on all hosts, however.

A.1.4 Component Types

After the global network parameters have been established, the individual component type definitions follow.

Self-Contained Components

Self-contained components are those which require no generation of code on the part of CoorSet, and are designed to be run as standalone programs. This instructs

CoorSet just to arrange for its instantiation at runtime, but generates no code for its interface. All communication through the associative interface is assumed to be manually-coded in components of this type. The syntax is:

- `execute` <integer> <Java identifier> [`include` <inclusion specification>]

The integer specifies the number of instances to start, and the class identifier is the name of the class containing the main method to start the component. The inclusion specification is optional, but you will find it useful to include some class files along with the bundle. **No** files are included by default with this specification, so you should at least indicate all class files needed by the class you are instantiating here.

The format for the inclusion specification is:

<inclusion specification> ::= '(' <inclusion entries> ')
<inclusion entries> ::= '(' <class name> ',' <path to class file> ')'
<class name> ::= '"' <Java identifier> '"'
<path to class file> ::= '"' <String> '"'

The <class name> must be a *fully qualified* Java class name, including the entire package name. No `import` statements take effect here. Do not simply indicate `MyApplication`, specify `MyPackage.MyApplication`. Also note that the name of the class, as well as the path to the class file, should both be enclosed in double quotation marks.

Firing Rules

Firing rules are a generalization of components in a data flow programming model, where rather than waiting on a single input, a firing rule waits on multiple inputs, possibly in a particular order, before producing its output. This requires specifying the rule, which is given by the following grammar:

```
<R> ::= <R> '<' <R> | <R> '||' <R> | <R> '&&' <R> | <ID>
<ID> ::= <String>
```

The '`<`' operator has the lowest precedence. In a rule of the form "`R1 < R2`", it specifies that `R1` must be satisfied, and then `R2` must be satisfied. Before `R1` is

satisfied, no checks are made for satisfaction of `R2`, and all events to satisfy `R2` must therefore occur after `R1` is satisfied.

The '`||`' operator has the next highest precedence. In a rule of the form "`R1 || R2`", it specifies that either `R1` or `R2` must be satisfied. Conditions for either's satisfaction is checked simultaneously. When one is satisfied, the rule fires with the appropriate data for the satisfied rule. (See section A.2 on the API for implementation details.) When both are satisfied, one of the two clauses is arbitrarily chosen.

The '`&&`' operator has the highest precedence. In a rule of the form "`R1 && R2`", it specifies that both `R1` and `R2` must be satisfied. When both are satisfied, the data from both is passed along.

The syntax for this type is:

- `rule ''`<R>`'' times` <integer> `execute` <Java method name> [`include` <inclusion specification>] `profile` <profile specification>
  - <R> is a rule specification as given by the grammar above.
  - `times` <integer> is the number of instances of this rule to create.
  - `execute` <Java class name> refers to a particular static method on a class to execute when the rule is satisfied, that will receive the data associated with the satisfied rules.
  - `include` <inclusion specification> specifies other class files to be included when the class is instantiated. It is optional, as indicated by the brackets. The format for this is:

    <inclusion specification> ::= '(' <inclusion entries> ')'
    <inclusion entries> ::= '(' <class name> ',' <path to class file> ')'
    <class name> ::= '"' <Java identifier> '"'
    <path to class file> ::= '"' <String> '"'

    The <class name> must be a *fully qualified* Java class name, including the entire package name. No `import` statements take effect here. Do not simply indicate `MyApplication`, specify `MyPackage.MyApplication`. Also note that the name of the class, as well as the path to the class file, should both be enclosed in double quotation marks.

The generated class is automatically included. Be sure to include at least the class that has the method to be executed.

- `profile` <profile specification> specifies the initial profile in the format:

  <profile specification> ::= '(' <profile entries> ')' | '(' ')'
  <profile entries> ::= <entry> | <entry> ',' <profile entries>
  <entry> ::= '(' <Attribute name> ',' <value> ')' | <Attribute name>

ID attributes are specified by a name alone. Valued attributes are parenthesized into a 2-tuple with the attribute name and its value.

Complex Component Types

Complex components have their initial interfaces specified in the configuration file. This definition gives the name of the component to be generated, the name of the class containing programmer-supplied computational code, the initial profile, and the initial accepts and requires interfaces. The syntax for this definition is:

`component` <integer> {
    `class` <Java class identifier>
    `target` <Java class identifier>
    [`execute` <Java method identifier>]
    [`include` <inclusion specification>]
    <zero or more profile, accepts, rule, or requests lines>
}

The `class` line specifies the Java class that contains the programmer code to be linked in. The `target` line specifies the class that will be generated as the component interface. The `execute` line, optional as indicated by the brackets, specifies a method to be executed in the target class as soon as the component comes online. If no method is selected, then the component will just listen on its accepts interface when it comes online.

The `include` line specifies other class files to be included when the class is instantiated. It is optional, as indicated by the brackets. The format for this is:

<inclusion specification> ::= '(' <inclusion entries> ')
<inclusion entries> ::= '(' <class name> ',' <path to class file> ')'
<class name> ::= `"` <Java identifier> `"`
<path to class file> ::= `"` <String> `"`

The <class name> must be a *fully qualified* Java class name, including the entire package name. No `import` statements take effect here. Do not simply indicate `MyApplication`, specify `MyPackage.MyApplication`. Also note that the name of the class, as well as the path to the class file, should both be enclosed in double quotation marks.

The automatically generated class, listed in the `class` line is automatically included. Be sure to include other classes, including the class listed in the `target` line in the inclusion statement, along with any other supporting classes.

Then zero or more lines specifying the profile, the accepts interface, and the requests interface follow. Each of these types is specified as follows:

- `profile` <profile spec>

  This specifies attributes in the initial profile. The profile specification is the same as that for firing rules, and is repeated here for completeness:

<profile spec>	::=	'(' <profile entries> ')' | '(' ')'
<profile entries>	::=	<entry> | <entry> ',' <profile entries>
<entry>	::=	<Attribute name> |
		'(' <Attribute name> ',' <Attribute value> ')'

  ID attributes are specified with a name alone. Valued attributes are parenthesized into a 2-tuple with the attribute name and its value.

  Multiple profile specifications may be given, and they will all be concatenated together to form the initial profile. Attributes appearing more than once in profile lines will end up getting the value specified by the last appearance of that attribute.

- `accepts ''`<transaction name>`''` <Java method name> `(` <type list> `)`

  This specifies a transaction to be accepted. The transaction name is given as the first argument in quotation marks, followed by the name of a method in the target class to handle that transaction. The signature of the transaction, given

as a list of the types of the transaction payload, is given last as a parenthesized list of Java type identifiers.

- `rule` <R> `execute` <Java method name>

  This is a multi-transactional entry in the accepts interface in the form of a firing rule. These rules behave like standalone firing rules, except they can be satisfied repeatedly. When a rule is satisfied, it is reset to its initial state for processing again. For firing rules that operate a limited number of times, the linked method name is passed as the first parameter the index of the satisfied rule; this can be used to delete the rule from the interface to keep it from being satisfied again. The method must take exactly two parameters; an integer and a `java.util.Hashtable`. The hash table contains the `Message` objects from the satisfied rule, keyed by the message type. It is important that for any possible satisfaction of the rule, a message type be listed only once; otherwise it is ambiguous which instance's message will be in the hash table.

- `requires` ``<transaction name>'' <Java method name> ``<selector>'' `(` <type list> `)`

  This specifies a transaction that is required. For historical reasons `requests` is acceptable as a synonym for `requires`. The transaction name is given as the first argument. The method name specifies a method to be generated in the component interface class that can be called by programmer code to make a request. This method is a shorthand for the general interface for making requests, described in section A.2. The initial selector for this request is given as the third argument in quotation marks, followed by a parenthesized, comma-delimited list of Java types representing the signature of the transaction.

A.2 Runtime System, API Description, and Current Status

The CoorSet language supplies the initial state of the system, as well as automatically generated interfaces for selected components. These interfaces provide the access to the communications medium, by way of sending and receiving messages. Each component type has its own interface.

The current implementation is deployed as a set of Java classes under JDK 1.4.1. When the Light-weight Reliable Multicast (LRMP) is chosen, all communication happens under a single multicast group and port via UDP packets, with reliability provided by LRMP. The facility for distributed application launching to start the components is provided by our Component Starting Component (CSC), which uses the Java Cryptographic Extensions to authenticate requests.

A.2.1 Supporting Classes Common to All Types

There are several classes that accompany the Associative Interface code, but only some of them will be directly used by programmers. These all belong to the `AssocBroadcast` package.

- Attribute

  The `Attribute` class is an abstract class, so you will never actually instantiate it, but rather will do so with one of the following subclasses. You then use these objects with `addAttrToProfile` to add attributes to the profile.

  - IDAttribute

    This attribute has no value associated with it, and functions like a flag. Selectors will test for the existence or absence of attributes of this type. The only method of this class is the constructor `IDAttribute(String s)`, which creates an object with the name `s`.

  - ValuedAttribute

    This attribute does have a value associated with it. Testing for the existence or absence of this absence is not provided. If a selector tests a valued attribute and that attribute is not present, by definition it will not match. The value has to be an actual instance of a class and *not* a primitive data type; however you can (and typically will) use the "wrapper" classes like `Integer` to use these types. Most often `Integer` or `String` objects are used.

    The constructor is `ValuedAttribute(String s, Comparable v)`, which creates an attribute with name `s` and value `v`. `v` must implement the interface `java.io.Comparable` because selectors do equality, less than, and greater than comparisons on these attributes.

- Selector

  The `Selector` class is also an abstract class, but unlike `Attribute` you will never directly use or instantiate its subclasses. You enclose these objects in `Message` objects to be their selectors. You obtain objects of this type by calling the static `Selector.generate(String s)`, where `s` contains a boolean expression which is analyzed by a built-in parser, and returns an appropriate Selector object.

  The grammar for Selectors is as follows. Strings enclosed in single quotes (‘’) represent literals, but the quotes themselves are not part of the literal.

```
<S> ::= id | id <op> val | <S> ‘&&’ <S> | <S> ‘||’ <S> |
        ‘(’<S>‘)’ | ‘!TRUE’ | ‘!FALSE’
<op> ::= ‘==’ | ‘<’ | ‘>’ | ‘<=’ | ‘>=’
```

  A selector is therefore one of the following:

  - The string ‘`!TRUE`’, representing a constant true (all recipients).
  - The string ‘`!FALSE`’, representing a constant false (no recipients).
  - A single flag attribute `id`, where `id` is the name of a profile entry (all recipients that have `id` in their profile).
  - A comparison `id <op> val`, where `id` is the name of a valued profile entry, `<op>` is one of the operators above which have the same meanings as they do in Java, and `val` is a value to compare that profile entry's value to.
  - A compound OR of two selectors of the form `<S1> || <S2>` (all recipients for whom either `<S1>` or `<S2>` is true).
  - A compound AND of two selectors of the form `<S1> && <S2>` (all recipients for whom both `<S1>` and `<S2>` are true).
  - A parenthesized selector of the form `(<S>)`. This has the same meaning as `<S>` by itself, but it most useful for proper grouping in compound expressions.

- Message

  The `Message` is the vector of all communication. Each message has five components:

  1. **A selector.** Acquired from the `Selector.generate` method above, the selector determines which components will and will not receive this message.
  2. **A transaction type.** This is an application-specific message type. These are defined and used by the application as it sees fit.
  3. **A protocol string.** Associative Broadcast will eventually support both asynchronous (data-flow) semantics where a message is broadcast and then processing continues, and synchronous (call-return) semantics where a message is broadcast a response is awaited before returning control. Currently, only asynchronous semantics are supported, and this parameter should always be `Message.p_ASYNC`.
  4. **A sender identification.** Just like the selector defines the recipients, this string identifies the source of the message.
  5. **An object array.** This is the payload. Its contents are defined and used as the applications sees fit.

  The constructor for a Message object is:

  - `public Message(Selector s, String transaction, Integer protocol, Object[] arguments)`
    - `s` is the selector can be generated from a textual string through the `Selector.generate` method described above.
    - `transaction` is a `String` with the transaction type.
    - `protocol` specifies whether this is an asynchronous message, meaning that the message is sent and then computation continues; or a synchronous message, meaning the interface should wait to receive a response. At present, only asynchronous messages are supported, so this argument should always be `Message.p_ASYNC`.

* `arguments` is the application-specific payload. Please note that all of these objects *must* implement the `java.io.Serializable` interface, so they can be transmitted over the network.

A.2.2 Self-Contained Component Interface

The self-contained component handles all of its own communications at the basic level, and therefore, it is responsible for maintaining its own interfaces. It has access to the basic messaging primitives for broadcasting, reception, and profile maintenance. These components will open an instance of the AssocBroadcast.AssociativeInterface class with the proper communication parameters. Since it is a self-contained component, CoorSet does not have a way to apply the system-wide network parameters. Therefore, any components of this type must be hardcoded with the same network parameters. The API for the AssociativeInterface class is as follows:

- `public Message receiveMessage()`

  This will return the first waiting message on the queue that matches the current profile. Note that the selectors of messages are matched against the local profile when this method is called; all messages are queued otherwise. If no message is waiting, this method will block until one is received.

  This method will throw a `java.io.IOException` if there is a problem with the reception, such as the interface being closed while waiting.

- `public AssociativeInterface(java.net.InetAddress mcastGroup, int port, int ttl, String id)`

  - `mcastGroup` contains the internet address of the multicast group used for communication.
  - `port` contains the UDP port used for communication.
  - `ttl` specifies the time-to-live for all datagrams broadcast from this interface; it must be large enough to span your entire computational network.
  - `id` is a unique `String` identifier used by the interface to mark the packets it broadcasts, so that it can ignore them. Semantics of associative

communication state that the broadcasting component is never in the receiver set.

The constructor will throw a `java.io.IOException` if it is unable to open the appropriate network ports.

- `public void close()`

  This closes the associative interface. Any current pending receive calls will terminate with exceptions, and the interface will no longer be available for use. To resume communication, a new instance must be created.

- `public void addAttrToProfile(Attribute a)`

  This adds an attribute to the local profile. The argument will be a subtype of `AssocBroadcast.Attribute`, which is either `AssocBroadcast.IDAttribute` for ID attributes, or `AssocBroadcast.ValuedAttribute` for valued attributes. If an attribute with the same name already exists in the profile, it is replaced with the argument.

- `public boolean removeAttrFromProfile(String symbol)`

  This removes the attribute from the local profile with the name `symbol`. The method returns `true` if the attribute was found and removed, and `false` if the attribute was not found.

- `public void printProfile(java.io.PrintStream out)`

  This prints the current profile to the `PrintStream` given as the argument.

- `public void broadcastMessage(Message m)`

  This broadcasts a message to the computational network. The `Message` object must be constructed with the proper selector, transaction type, protocol, and application payload. This is all accomplished through the Message constructor. See its section for proper syntax.

A.2.3 Firing Rule Interface

The firing rule interface is at present incomplete, and we recommend against its use for now.

A.2.4 Complex Component Interface

In a complex component, communication is not handled directly through an associative interface, but rather a subclass of `CoorSet.ComplexNode.Interface`. In these component types, the accepts interface handles all message reception, so no reception primitive is provided through the API. The methods for broadcasting and profile maintenance are the same as those for the associative interface. The entire API is as follows:

- `public void addRequest(String transName, String selectorStr)`

  This adds a request to the requests interface. This does not actually generate a request, but adds it to the table for later invocation. If an entry already exists in the table, then the selector is updated to the given argument.

  - `transName` is the transaction identifier.
  - `selectorStr` is a string representation of the selector. It is converted internally to a `Selector` object.

- `public boolean delRequest(String transName)`

  This removes the request from the requests interface with the transaction name `transName`.

- `public boolean setSelector(String transName, String inSelector)`

  This changes the selector of an entry in the requests interface with the name `transName` to have the selector `inSelector`. This method differs from the function of `addRequest` in that if the entry does not already exist in the requests interface, it will *not* be added. It will return `false` in this case or if the selector is invalid, and `true` if the entry was found and the selector was updated.

- `public boolean addAccept(String transName, String methodName, Class[] paramTypes)`

  This adds an entry to the accepts interface.

  - `transName` is the name of the transaction to accept.

- `methodName` is the name of a method on the target class to invoke. At this time, only methods on the designated target class may be referenced at this point.
- `paramTypes` is the list of types of the parameters for this request. These parameters *must* match the signature of the method to be called.

This method returns `true` if the addition was successful. It returns `false` if there was an error, such as the method not being found, or the argument types mismatching.

- `public boolean delAccept(String transName)`

  This deletes the entry from the accepts interface with the transaction name `transName`.

- `public void request(String transName, java.util.List parameters)`

  This generates a request with transaction name `transName` with its currently set selector and broadcasts it over the network. `parameters` is any kind of `java.util.List` containing the parameters to be placed in the payload. All methods created in the CoorSet definition for the requests interface are just wrapped calls to this method with the appropriate transaction name.

- `public void broadcastMessage(AssocBroadcast.Message m)`

  Deprecated. This broadcasts a Message object to the network. Its use is discouraged for complex components of this type; use the requests interface instead.

- `public String get_id()`

  This returns the unique identifier of this interface.

- `public void addAttrToProfile(AssocBroadcast.Attribute a)`

  This adds an attribute to the local profile. The argument will be a subtype of `AssocBroadcast.Attribute`, which is either `AssocBroadcast.IDAttribute` for ID attributes, or `AssocBroadcast.ValuedAttribute` for valued attributes. If an attribute with the same name already exists in the profile, it is replaced with the argument.

- `public boolean removeAttrFromProfile(String symbol)`

  This removes the attribute from the local profile with the name `symbol`. The method returns `true` if the attribute was found and removed, and `false` if the attribute was not found.

- `public void printProfile(java.io.PrintStream out)`

  This prints the current profile to the `PrintStream` given as the argument.

A.3 The Component Starting Component (CSC)

A key part of the runtime system is the "Component Starting Component", which forms the infrastructure for starting components. These are started *manually* on hosts within the same multicast domain. They receive requests to start components, bundled with the class data for those components, and fork them off in new threads. Although you should not need to use the API for the CSC, you will need to know its basic operation so that you can generate security keys and boot them up on stations in your network.

A.3.1 The Key Generator

For security reasons, all requests must be signed with a trusted Digital Signature Algorithm (DSA) private key. You will need a `java.security.KeyPair` object containing a DSA public/private key pair for use both with the CSC, and the CSC client classes. A key generator class, `CoorSet.CSC.Keygen`, has been provided for this purpose. Run it as follows:

```
java CoorSet.CSC.Keygen
```

It will prompt you: "`Target filename:` " Enter a pathname in which to save the key pair. There will then be a delay as it generates the keys, and then it will terminate. You will then have a keypair in the filename specified. For the rest of this section, we shall assume the filename is `keys`.

A.3.2 The Component Starting Component

Once you have a keypair, you are ready to start any number of Component Starting Components. At present, all CSCs and clients must use the same keypair. In the future, the choice of the DSA algorithm will allow us to use different keys and maintain lists of authorized public keys. The CSC comes precompiled with a default multicast group, port number, and time-to-live values, but alternate values can be chosen. Start the CSC as follows:

```
java CoorSet.CSC.CSC <keypair file> [<multicast group> <port> <time
to live>]
```

The last three parameters are optional, in which case the CSC defaults to compiled-in defaults; at the time of this writing, that is the group 229.1.1.1, port 10000, with a time-to-live of 20.
For example:

```
java CoorSet.CSC.CSC keys 230.1.2.3 23456 5
```

This specifies that the key pair is in the file `keys`, the multicast group is 230.1.2.3, the port is 23456, and the time-to-live is 5.

A.4 Directory Server for United Devices

If you are using the United Devices method, instead of using the CSC, you will instead start a Directory Server in a well-known location.

Execute the following command to start the Directory Server:

```
java AssocBroadcast.DS.DirectoryServer <port>
```

where `<port>` is the same as given in the `dsport` directive.

Appendix B

CapCoorSet

We give here some supplementary material on CapCoorSet. The structure of the language derives from CoorSet, but has been improved both for clarity and for the additional information required by CapCoorSet. In section B.1, the Broker definition is given to illustrate the language. In section B.2, the step-by-step workflow of a transaction in the example stock market is given.

B.1 Condensed Broker Definition

We have condensed the definition to omit some minor implementation details to save space. The first two lines give the name of the interface class to be automatically generated and the programmer-provided class, respectively. The optional `execute` line specifies a method in the programmer-provided class to be executed as soon as the service comes online; if this is omitted, the service just waits for requests. The `profile` line gives the profile for the entire service, which in this case consists of only one attribute; a comma-delimited list here would specify more.

The two `accepts` lines add entries to the accepts interface. Each first lists the human-readable name of the operation, the method on the programmer-supplied class to be invoked to service a request, the parameters that method accepts, and finally another method on the programmer-supplied class which will be invoked to decide the result of a handshake for that operation, when one is requested.

The two `requires` lines add entries to the requires interface. Each first lists the human-readable name of the operation, the default selector for discovering

```
service {
 generate class Broker
 computation class BrokerLib
 execute startup
 profile ("Broker")

 accepts "QuoteRequest" invokes quoteReq parameters (String)
   handshaker hsQuoteReq
 accepts "PurchaseAtQuote" invokes purchase parameters (Integer,
   Check) handshaker hsPurchase

 requires "CheckRequest" with selector "Bank" parameters (Integer,
   java.security.cert.Certificate)
 requires "QuoteRequest" with selector "Issuer" parameters ()
}
```

Figure B.1: Broker condensed definition in CapCoorSet language

matching services, and the parameters that operation requires. Although other services are not shown here, the parameters for a given requests interface must match a corresponding accepts interface in addition to the selector; either being false is not a match.

B.2 Stock Example Workflow

We present the workflow for the system in figure B.2. The numbering of the steps in the figure correspond to the list following. For clarity, we omit return values that merely indicate just a successful invocation, such as when a Check is redeemed in steps 8 and 12. We also omit the opening of accounts, and just present the interaction when stock is quoted and purchased. The operation of the system is as follows:

1. A Customer initiates a stock purchase. The customer begins discovery with selector "Broker" which will locate at least one broker. The customer randomly selects one and handshakes for a QuoteRequest capability. Here we see one of the many times the *initial issuance mechanism* is used. This capability is given freely by the broker, with no contractual limitation. The customer

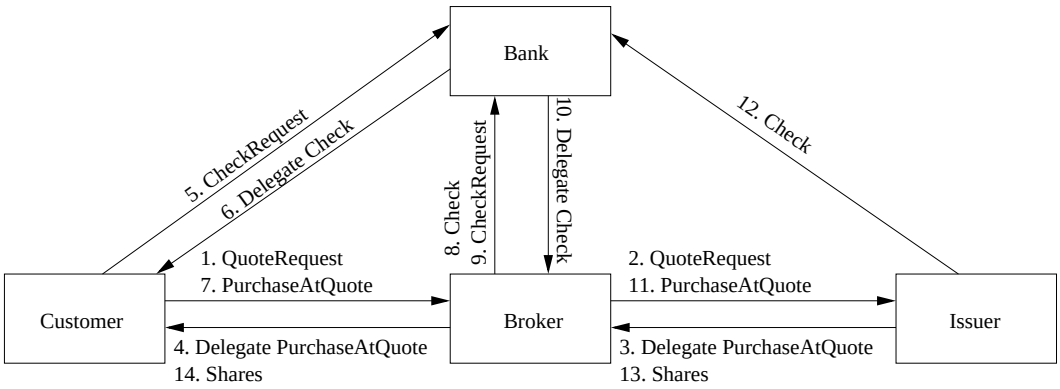

Figure B.2: Stock Market Example

then invokes the QuoteRequest capability, with the four-letter stock symbol as parameter. For this example, let us assume the symbol is "UTCS."

2. This invocation causes the broker to use the discovery protocol with the selector "Issuer && Symbol == UTCS" to find an issuer that supplies the desired stock. The handshake protocol is then invoked on the discovered issuer to request a QuoteRequest capability on the issuer. This capability is given freely without restriction by the issuer. The broker then invokes this capability on the issuer to get the current quote.
3. For this example, the issuer decides a random share price P_I on the interval $[50, 80]$. The issuer then generates a PurchaseAtQuote capability for that price, but restricted with a contract that allows its use only for a limited amount of time. Here we see one of the *contractually-limited capabilities*. In this example, we limit purchase quotes to five minutes from time of issue. Since the clock used to generate the timestamp is the same clock that will be later used to check it, there is no time synchronization issue. The broker then stores this capability in a local table.
4. The broker then adds a percentage onto the quoted price, 5% in this example, and generates its own PurchaseAtQuote capability for the customer with that price $P_I \times 1.05$, which we will call P_B, and a stored pointer to the capability remembered in the previous step.[1] The broker also time-limits this capability

[1]The PurchaseAtQuote capability from the Issuer could be stored encrypted in the Pur-

to be slightly less than the validity period on the PurchaseAtQuote capability provided by the issuer, to ensure it does not expire before the capability provided to the customer. Note that this capability is not limited to any particular customer, so one could pass it to another and take advantage of the same quoted stock price. Here is one instance of *delegation of authorizations*, as well as *scalability* in keeping the contracts stored by the client.

5. Should the customer decide to purchase stock at that price, and in our example the customer always does, the customer then needs to get a Check from the bank to cover the cost, and then request the share purchase from the broker with that check enclosed. The check is another *delegation of authorization*: to draw funds from the Customer's account. The customer decides on a number of shares n to purchase. In our example, the customer chooses n randomly from the interval $[1, 50]$. The customer then invokes the previously acquired CheckRequest capability on the bank, providing the identity of the broker from the PurchaseAtQuote capability and the amount $n \times P_B$ required to cover the purchase request.
6. Assuming the customer has sufficient funds, the bank deducts those funds from the balance, and issues a Check capability to the customer, limited with a contract to be redeemable only once and only by the intended recipient (in this case, the broker).
7. The customer then invokes the PurchaseAtQuote capability, providing the desired number of shares n as well as the Check.
8. The broker invokes the Check, which is itself a capability usable only once on the bank to redeem funds to the recipient's account.
9. Assuming the Check is valid, the broker then invokes its CheckRequest capability on the bank to write a check to the issuer for the amount $n \times P_I$ required to cover the purchase.
10. After depositing the customer's check, we are certain the Broker has sufficient funds, and so the bank provides a Check capability for that amount, limited by contract to the issuer and only to one use.
11. The broker then invokes the PurchaseAtQuote capability on the issuer, pro-

chaseAtQuote capability delegated to the Customer, removing the need for the Broker to store any state, reducing its space requirement further. For simplicity, we just let the Broker remember the capability received from the Issuer.

viding the desired number of shares n as well as the Check.

12. The issuer invokes its Check to deposit the funds into its account.
13. The issuer then issues the Shares, which is an application-specific object representing a number of shares, signed cryptographically by the issuer. This certificate is then returned to the broker as the result of the PurchaseAtQuote invocation.
14. This certificate is then returned to the customer by the broker, as the result of the customer's PurchaseAtQuote invocation. The interaction is now complete.

Appendix C

Verification Logic Soundness Proof

Given that past- and future-time temporal logic is sound, we show the given logic is sound. Let $\Phi \vdash \phi$ denote that given premises Φ, formula ϕ is derivable from Φ in the logic. Let $\Phi \propto \phi$ denote that Φ semantically entails ϕ. Note that traditionally, $\models$ is used to denote semantic entailment, but the logic uses $\models$ in the style of temporal logic to indicate what is true in a given global state. To avoid confusion, we use $\propto$ to denote the semantic entailment of a conclusion by a set of premises.

Soundness Theorem. $\Phi \vdash \phi \rightarrow \Phi \propto \phi$.

Proof. We prove by induction on the length of a derivation.

Base cases.

Handshake axiom. Axiom 7.1 states that the knowledge set of a service at any state can contain the handshake capability on any service. We must show that:

$$\gamma_i \models \overline{s_\emptyset[\top]} \in K_c \propto \gamma_i \models \overline{s_\emptyset[\top]} \in K_c$$

By definition of service-oriented systems, a service can always construct the capability to the handshake operation of another service.

Capability axiom. Axiom 7.2 states that a service knows any capability with any

contract for any operation it provides. We must show:

$$\gamma_i \models \overline{s_k[C]} \in K_s \propto \gamma_i \models \overline{s_k[C]} \in K_s$$

By definition, the service which provides an operation can create a capability with any contract to that operation. It is the only one that can.

Trivial contract axiom. Axiom 7.12 states that any service believes the trivial contract. We must show:

$$\gamma_i \models s \models\!\equiv \top \propto \gamma_i \models s \models\!\equiv \top$$

By semantic definition of system operation, the trivial contract is always true and always believed.

Inductive step. Suppose that $\Phi \vdash \phi$, and assume that $\Phi \propto \phi$. We now show that for the possible choices of ϕ', if $\Phi \cup \phi \vdash \phi'$, then $\Phi \cup \phi \propto \phi'$. We examine each possible ϕ' by considering each rule of inference.

Connectedness rule. Suppose $\Phi \vdash \gamma_i \models \Diamond c \models\!\equiv s \mid\!\sim (u, m)$ with last rule 7.3. This implies $\Phi \vdash \gamma_{i-1} \models s \mid\!\sim (c, m)$ and $\Phi \vdash \gamma_{i-1} \models \texttt{connected}(s, c)$. By the induction hypothesis, $\Phi \propto \gamma_{i-1} \models s \mid\!\sim (c, m)$ and $\Phi \propto \gamma_{i-1} \models \texttt{connected}(s, c)$. The semantic definition of our system states that two services which are connected can send messages to one another, and so this implies $\Phi \propto \Diamond c \models\!\equiv s \mid\!\sim (u, m)$.

Says-belief rule. Suppose $\Phi \vdash \gamma_i \models u \models\!\equiv s \models\!\equiv f$ with last rule 7.4. This implies $\Phi \vdash \gamma_i \models u \models\!\equiv s \mid\!\sim (u, s \models\!\equiv f)$. By the induction hypothesis, it holds that $\Phi \propto \gamma_i \models u \models\!\equiv s \mid\!\sim (u, s \models\!\equiv f)$. We have assumed that channels are free from tampering and identity is assured, and therefore u knows s sent this message. u can therefore conclude that s believes f, as s is attesting to that fact. Therefore, $\Phi \propto u \models\!\equiv s \models\!\equiv f$.

Change of frame rule. Suppose $\Phi \vdash \gamma_i \models s \models\!\equiv \diamond u \mid\!\sim (v, m)$ with last rule 7.5. This implies that for some previous state γ_p where $p < i$, $\Phi \vdash \gamma_p \models \Diamond s \models\!\equiv u \mid\!\sim (v, m)$. By the induction hypothesis, $\Phi \propto \gamma_p \models \Diamond s \models\!\equiv u \mid\!\sim (v, m)$. By the definition of $\Diamond$, there exists a future state γ_k $(k > p)$ where $s \models\!\equiv u \mid\!\sim (v, m)$. By the definition of $\diamond$, there exists another future state γ_i $(i > k)$ where $s \models\!\equiv \diamond u \mid\!\sim (v, m)$. Therefore, $\Phi \propto \gamma_i \models s \models\!\equiv \diamond u \mid\!\sim (v, m)$.

Sending rule. Suppose $\Phi \vdash \gamma_i \models c \mathrel{|\!\sim} (s, m)$ with last rule 7.6. This implies $\Phi \vdash \gamma_i \models m \in K_c$ and $\Phi \vdash \gamma_i \models c \overrightarrow{\mathrel{|\!\sim}} (s, m)$. By the induction hypothesis, $\Phi \propto \gamma_i \models m \in K_c$ and $\Phi \propto \gamma_i \models c \overrightarrow{\mathrel{|\!\sim}} (s, m)$. The definition of our system model states that if a service c knows a message m and it intends to send that m, it can send that message to any other service s, though it is not guaranteed to be received. This implies that $\Phi \propto \gamma_i \models c \mathrel{|\!\sim} (s, m)$.

Send-belief rule. Suppose $\Phi \vdash \gamma_i \models c \mathrel{|\!\sim} (s, c \mathrel{|\!\equiv} f)$ with last rule 7.7. This implies $\Phi \vdash \gamma_i \models c \mathrel{|\!\equiv} f$ and $\Phi \vdash \gamma_i \models c \overrightarrow{\mathrel{|\!\sim}} (s, c \mathrel{|\!\equiv} f)$. By the induction hypothesis, $\Phi \propto \gamma_i \models c \mathrel{|\!\equiv} f$ and $\Phi \propto \gamma_i \models c \overrightarrow{\mathrel{|\!\sim}} (s, c \mathrel{|\!\equiv} f)$. The definition of our system model states that if a service c knows a message, which in this case is that c believes a formula f, and it intends to send that m, it can send that message to any other service s. This implies that $\Phi \propto \gamma_i \models c \mathrel{|\!\sim} (s, c \mathrel{|\!\equiv} f)$.

Handshake rule. Suppose $\Phi \vdash \gamma_i \models s \mathrel{|\!\sim} (c, \overline{s_k[\mathcal{C}]})$ with last rule 7.8. This implies that for some previous state γ_p where $p < i$, $\Phi \vdash \gamma_p \models s \mathrel{|\!\equiv} c \triangleright \overline{s_k[\mathcal{C}]}$ and $\Phi \vdash \gamma_p \models s \mathrel{|\!\equiv} \diamond c \mathrel{|\!\sim} (s, \overline{s_\emptyset[\top]}(s_k))$. By the induction hypothesis, this implies $\Phi \propto \gamma_p \models s \mathrel{|\!\equiv} c \triangleright \overline{s_k[\mathcal{C}]}$ and $\Phi \propto \gamma_p \models s \mathrel{|\!\equiv} \diamond c \mathrel{|\!\sim} (s, \overline{s_\emptyset[\top]}(s_k))$. This means that s believes that c invoked its handshake capability asking for a capability to operation s_k, and s also believes that c is entitled to a capability for operation s_k with contract $\mathcal{C}$. By definition of a service's behavior, if it is asked for a capability to an operation through a handshake and believes the asking client should have one, it will issue one to that client. It will intend to send that message, and due to the sending rule 7.6 and the capability axiom 7.2, it sends that message. Therefore, $\Phi \propto \gamma_i \models s \mathrel{|\!\sim} (c, \overline{s_k[\mathcal{C}]})$.

Delegation rule. Suppose $\Phi \vdash \gamma_i \models \overline{s_k[\mathcal{C}]} \in K_c$ with last rule 7.9. This implies that $\Phi \vdash \gamma_i \models c \mathrel{|\!\equiv} \diamond s \mathrel{|\!\sim} (c, \overline{s_k[\mathcal{C}]})$. By the induction hypothesis, $\Phi \propto \gamma_i \models c \mathrel{|\!\equiv} \diamond s \mathrel{|\!\sim} (c, \overline{s_k[\mathcal{C}]})$. Since c believes s said this, c clearly received the message, and now has $\overline{s_k[\mathcal{C}]}$ available to it, and therefore it is committed to c's knowledge set. Therefore, $\Phi \propto \gamma_i \models \overline{s_k[\mathcal{C}]}$.

Invocation rule. Suppose $\Phi \vdash \gamma_i \models !s_k(P)$ with last rule 7.10. This implies that for some previous state γ_p where $p < i$, $\Phi \vdash \gamma_p \models s \mathrel{|\!\equiv} \diamond c \mathrel{|\!\sim} (s, \overline{s_k[\mathcal{C}]}(P))$ and $\Phi \vdash \gamma_p \models s \mathrel{|\!\equiv} \mathcal{C}$. By the induction hypothesis, $\Phi \propto \gamma_p \models s \mathrel{|\!\equiv} \diamond c \mathrel{|\!\sim} (s, \overline{s_k[\mathcal{C}]}(P))$ and $\Phi \propto \gamma_p \models s \mathrel{|\!\equiv} \mathcal{C}$. By the definition of a service, if a service s is presented

with a valid capability to an operation s_k with contract $\mathcal{C}$, and s believes $\mathcal{C}$ to be true, then it executes the operation. Therefore, $\Phi \propto \gamma_i \models !s_k(P)$.

Result rule. Suppose $\Phi \vdash \gamma_i \models s_k(P) \in K_s$ with last rule 7.11. This implies that for the previous state γ_{i-1}, $\Phi \vdash \gamma_{i-1} \models !s_k(P)$. By the induction hypothesis, $\Phi \propto \gamma_{i-1} \models !s_k(P)$. By the semantic definition of the operation of a service, once it executes an operation with a set of parameters P, the result of that operation is available to it after that execution, which is the next state γ_i. Therefore, $\Phi \propto \gamma_i \models s_k(P) \in K_s$.

Jurisdiction rule. Suppose $\Phi \vdash \gamma_i \models c \mid\!\equiv f$ with last rule 7.13. This implies $\Phi \vdash \gamma_i \models c \mid\!\equiv s \Rightarrow f$ and $\Phi \vdash \gamma_i \models c \mid\!\equiv s \mid\!\equiv f$. By the induction hypothesis, $\Phi \propto \gamma_i \models c \mid\!\equiv s \Rightarrow f$ and $\Phi \propto \gamma_i \models c \mid\!\equiv s \mid\!\equiv f$. By the definition of the jurisdiction operator $\Rightarrow$, if c believes s has jurisdiction over a formula f, and c believes s believes f, then c will believe f. Therefore, $\Phi \propto \gamma_i \models c \mid\!\equiv f$.

□

Bibliography

[1] Martín Abadi and Bruno Blanchet. Analyzing security protocols with secrecy types and logic programs. *J. ACM*, 52(1):102–146, 2005.

[2] Martín Abadi, Bruno Blanchet, and Cédric Fournet. Just Fast Keying in the Pi Calculus. In *Programming Languages and Systems: 13th European Symposium on Programming (ESOP 2004), Lecture Notes in Computer Science 2986*, pages 340–354, Berlin, Germany, 2004. Springer-Verlag.

[3] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: the spi calculus. In *CCS '97: Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 36–47, New York, NY, USA, 1997. ACM Press.

[4] Martín Abadi, Edward Wobber, Michael Burrows, and Butler Lampson. Authentication in the Taos operating system. In *Proceedings of the 14th ACM Symposium on Operating System Principles*, pages 256–269, The Grove Park Inn and Country Club, Asheville, NC, 1993. ACM Press.

[5] Luís Filipe A. Andrade and José Luiz L. Fiadeiro. Interconnecting objects via contracts. In *UML'99 – Beyond the Standard (LNCS 1723)*, pages 566–583. Springer-Verlag, 1999.

[6] Jean-Marc Andreoli and Remo Pareschi. Linear Objects: logical processes with built-in inheritance. In *Proceedings of the 7th International Conference on Logic Programming*, pages 495–510, May 1990.

[7] Tony Andrews, Francisco Curbera, Hitesh Dholakia, Yaron Goland, Johannes Klein, Frank Leymann, Kevin Liu, Dieter Roller, Doug Smith, Satish Thatte,

Ivana Trickovic, and Sanjiva Weerawarana. Business Process Execution Languages for Web Services (BPEL4WS). `ftp://www6.software.ibm.com/software/developer/library/ws-bpel.pdf`, May 2003.

[8] Farhad Arbab, Marcello M. Bonsangue, and Frank S. de Boer. A coordination language for mobile components. In *SAC '00: Proceedings of the 2000 ACM Symposium on Applied Computing*, pages 166–173, New York, NY, USA, 2000. ACM Press.

[9] Farhad Arbab, Ivan Herman, and P. Spilling. An overview of Manifold and its implementation. *Concurrency: Practice and Experience*, 5(1):23–70, 1993.

[10] William A. Arbaugh, Narendar Shankar, and Y. C. Justin Wan. Your 802.11 wireless network has no clothes. *IEEE Wireless Communications*, 9(6):44–51, Dec 2002.

[11] Bryan Bayerdorffer. *Associative Broadcast and the Communication Semantics of Naming in Concurrent Systems*. PhD thesis, University of Texas at Austin, 1993.

[12] Bryan Bayerdorffer. Distributed computing with associative broadcast. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, 1995.

[13] Eshwar Belani, Amin Vahdat, Thomas Anderson, and Michael Dahlin. The CRISIS wide area security architecture. In *USENIX Security Symposium*, pages 15–30, Jan 1998.

[14] Giampaolo Bella and Lawrence C. Paulson. Mechanising BAN Kerberos by the inductive method. In *Computer Aided Verification*, pages 416–427, 1998.

[15] Naouel Ben Salem, Jean-Pierre Hubaux, and Markus Jakobsson. Reputation-based Wi-Fi Deployment. *The Mobile Computing and Communications Review (MC2R)*, 9(3), 2005.

[16] Massimo Benerecetti and Fausto Giunchiglia. Model checking security protocols using a logic of belief. In Susanne Graf and Michael I. Schwartzbach,

editors, *6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS2000), LNCS 1785*, pages 519–534, Mar 2000.

[17] Massimo Benerecetti, Fausto Giunchiglia, Maurizio Panti, and Luca Spalazzi. A logic of belief and a model checking algorithm for security protocols. In *13th FORTE/20th PSTV, IFIP TC6 WG6.1 joint international conference*, pages 393–408, Oct 2000.

[18] Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, Feb 2003.

[19] Lorenzo Bettini, Rocco De Nicola, and Rosario Pugliese. Klava: a Java framework for distributed and mobile applications. *Software – Practice and Experience*, 32:1365–1394, 2002.

[20] Dietrich Birngruber. CoML: Yet another, but simple component composition language. In *Proceedings of the Workshop on Composition Languages*, 2001.

[21] Andrea Bittau, Mark Handley, and Joshua Lackey. The final nail in WEP's coffin. `http://tapir.cs.ucl.ac.uk/bittau-wep.pdf`.

[22] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. The KeyNote trust-management system, version 2. IETF RFC 2704, 1999.

[23] Matt Blaze, Joan Feigenbaum, John Ioannidis, and Angelos D. Keromytis. *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, chapter The Role of Trust Management in Distributed Systems Security, pages 185–210. Lecture Notes in Computer Science State-of-the-Art series. Springer-Verlag, 2001.

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

[25] M. Boasson. Control system software. *IEEE Transactions on Automatic Control*, 38(7):1094–1106, Jul 1993.

[26] William Earl Boebert. On the inability of an unmodified capability machine to enforce the *-property. In *Proceedings of the 7th DoD/NBS Computer Security Conference*, pages 291–293, Sep 1984.

[27] Marcello M. Bonsangue, Joost N. Kok, and G. Zavattaro. Comparing coordination models based on shared distributed replicated data. In *SAC '99: Proceedings of the 1999 ACM Symposium on Applied Computing*, pages 156–165, New York, NY, USA, 1999. ACM Press.

[28] Nikita Borisov, Ian Goldberg, and David Wagner. Intercepting mobile communications: The insecurity of 802.11. In *The Seventh Annual International Conference on Mobile Computing and Networking (MOBICOM)*, Jun 2001.

[29] James C. Browne, Emery D. Berger, and A. Dube. Compositional development of performance models in POEMS. *International Journal of High Performance Computing Applications*, 14(4):283–291, Nov 2000.

[30] James C. Browne, Kevin Kane, and Hongxia Tian. An associative broadcast based coordination model for distributed processes. In *Coordination Models and Languages, Proceedings of COORDINATION 2002 (LNCS 2315)*, pages 96–110. Springer-Verlag, Apr 2002.

[31] Sonja Buchegger and Jean-Yves Le Boudec. Performance analysis of the CONFIDANT protocol (Cooperation Of Nodes: Fairness In Dynamic Ad-hoc NeTworks). In *Proceedings of the 3rd ACM International Symposium on Mobile Ad Hoc Networking & Computing*, pages 226–236, 2002.

[32] Sonja Buchegger and Jean-Yves Le Boudec. A robust reputation system for P2P and mobile ad-hoc networks. In *Proceedings of the Second Workshop on the Economics of Peer-to-Peer Systems*, 2004.

[33] Michael Burrows, Martín Abadi, and Roger Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, Feb 1990.

[34] I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proceedings of the 12th IEEE Computer Security Foundations Workshop (CSFW'99)*, pages 55–69, Los Alamitos, California, 1999. IEEE Computer Society.

[35] D. Chazan and W. L. Miranker. Chaotic relaxation. *Linear Algebra Applications*, 2(2):199–222, Apr 1969.

[36] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for web applications. *World Wide Web Journal*, 2:706–734, 1997.

[37] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[38] Bram Cohen. Incentives build robustness in BitTorrent. `http://www.bittorrent.com/bittorrentecon.pdf`, May 2003.

[39] Ellis Cohen and David Jefferson. Protection in the Hydra operating system. In *Proceedings of the Fifth Symposium on Operating Systems Principles*, pages 141–160, Nov 1975.

[40] Carlos E. Cuesta, Pablo de la Fuente, Manuel Barrio-Solórzano, and M. Encarnación Beato. Coordination in a reflective arcthitecture description language. In *Coordination Models and Languages, Proceedings of COORDINATION 2002 (LNCS 2315)*, pages 141–148. Springer-Verlag, 2002.

[41] Anupam Datta, Ante Derek, John C. Mitchell, and Dusko Pavlovic. A derivation system and compositional logic for security protocols. *Journal of Computer Security*, 13:423–482, 2005.

[42] Anupam Datta, John C. Mitchell, and Dusko Pavlovic. Derivation of the JFK protocol. Technical Report KES.U.02.03, Kestrel Institute, Jul 2002.

[43] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. Technical Report MIT/LCS/TR-23, MIT, 1965.

[44] Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Access control: principles and solutions. *Software: Practice and Experience*, 33(5):397–421, Apr 2003.

[45] T. Dierks and C. Allen. The TLS protocol, version 1.0 (RFC 2246). `http://www.ietf.org/rfc/rfc2246.txt`, Jan 1999.

[46] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, 1987.

[47] Danny Dolev and Dalia Malki. On distributed algorithms in a broadcast domain. In *International Conference on Automata, Languages, and Programming*, pages 371–387, 1993.

[48] John R. Douceur. The Sybil attack. In *Proceedings of the First International Workshop on Peer-to-Peer Systems (IPTPS '02)*, Mar 2002.

[49] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, 1988.

[50] E. Allen Emerson. *Handbook of Theoretical Computer Science*, volume B, chapter Temporal and modal logic, pages 955–1072. MIT Press, 1990.

[51] Michael Feldman, John Chuang, Ion Stoica, and Scott Shenker. Hidden-action in multi-hop routing. In *ACM E-Commerce Conference (EC'05)*, Jun 2005.

[52] Stephan Flake and Wolfgang Mueller. Past- and future-oriented time-bounded temporal properties with OCL. In *2nd International Conference on Software Engineering and Formal Methods (SEFM 2004)*, pages 154–163. IEEE Computer Society Press, Sep 2004.

[53] Ian Foster and Carl Kesselman, editors. *The Grid 2: Blueprint for a New Computing Infrastructure*. The Morgan Kaufmann Series in Computer Architecture. Morgan Kaufmann, 2nd edition, 2004.

[54] Ian Foster, Carl Kesselman, and Steven Tuecke. The anatomy of the grid: Enabling scalable virtual organizations. *Int. J. High Perform. Comput. Appl.*, 15(3):200–222, 2001.

[55] Eric Freudenthal, Tracy Pesin, Lawrence Port, Edward Keenan, and Vijay Karamcheti. dRBAC: Distributed role-based access control for dynamic coalition environments. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS 2002)*, Jul 2002.

[56] Eric J. Friedman and Paul Resnick. The social cost of cheap pseudonyms. *Journal of Economics and Management Strategy*, 10(2):173–199, 2001.

[57] David Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[58] Holger Giese. Contract-based component system design. Technical Report 2399, Institut für Informatik, Westfälische Wilhelms-Universität, Jan 2000.

[59] Globus Alliance. Open Grid Services Architecture. `http://www.globus.org/ogsa/`.

[60] Li Gong. A secure identity-based capability system. In *Proceedings of the 1989 IEEE Symposium on Security and Privacy*, pages 56–63, May 1989.

[61] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann Publishers, Inc., San Francisco, CA, USA, 1993.

[62] Koji Hasebe and Mitsuhiro Okada. A logical verification method for security protocols based on linear logic and BAN logic. In *Mext-NSF-JSPS International Symposium, ISSS 2002 (LNCS 2609)*, pages 417–440, Nov 2002.

[63] Klaus Havelund and Grigore Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24:189–215, 2004.

[64] Joseph Henrich. Cooperation, punishment, and the evolution of human institutions. *Science*, 312(5770):60–61, Apr 2006.

[65] Gerard J. Holzmann. *The SPIN Model Checker, Primer and Reference Manual*. Addison-Wesley, Boston, MA, USA, 2004.

[66] IEEE 802.11 Working Group. IEEE 802 part 11: Wireless LAN medium access control (MAC) and physical layer (PHY) specifications. `http://standards.ieee.org/getieee802/802.11.html`, 1997.

[67] Audun Jøsang. A subjective metric of authentication. In *Proceedings of the 5th European Symposium on Research in Computer Security (ESORICS'98)*, 1998.

[68] Audun Jøsang, Dieter Gollmann, and Richard Au. A method for access authorisation through delegation networks. In *Australasian Information Security Workshop*, 2006.

[69] Audun Jøsang, Elizabeth Gray, and Michael Kinateder. Simplification and analysis of transitive trust networks. *Web Intelligence and Agent Systems Journal*, 4(2):139–161, 2006.

[70] Audun Jøsang and Simon Pope. Semantic constraints for trust transitivity. In S. Hartmann and M. Stumptner, editors, *Proceedings of the Asia-Pacific Conference of Conceptual Modelling (APCCM)*, Feb 2005.

[71] Richard Y. Kain and Carl E. Landwehr. On access checking in capability-based systems. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy*, pages 95–100, 1986.

[72] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The EigenTrust algorithm for reputation management in P2P networks. In *Proceedings of the Twelfth International World Wide Web Conference*, 2003.

[73] Kevin Kane and James C. Browne. The Component Starting Component: an environment for distributed systems and peer to peer research. Technical Report TR-03-42, Department of Computer Sciences, University of Texas at Austin, 2003.

[74] Kevin Kane and James C. Browne. CoorSet: A development environment for associatively coordinated components. In *Coordination Models and Languages, Proceedings of COORDINATION 2004 (LNCS 2949)*, pages 216–231. Springer-Verlag, Feb 2004.

[75] Kevin Kane and James C. Browne. On classifying access control implementations for distributed systems. In *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies (SACMAT'06)*, pages 29–38, Jun 2006.

[76] Kevin Kane and James C. Browne. Using uncertainty in reputation methods to enforce cooperation in ad-hoc networks. In *Proceedings of the ACM Workshop on Wireless Security (WiSe 2006)*, pages 105–113, Sep 2006.

[77] Alan H. Karp, Guillermo J. Rozas, Arindam Banerji, and Rajiv Gupta. Using split capabilities for access control. *IEEE Software*, 20(1):42–49, Jan 2003.

[78] Daniel Kraft and Günter Schäfer. Distributed access control for consumer operated mobile ad-hoc networks. In *Proceedings of the First IEEE Consumer Communications and Networking Conference (CCNC'2004)*, Jan 2004.

[79] Timo Kyntaja. A logic of authentication by Burrows, Abadi, and Needham. `http://www.tml.tkk.fi/Opinnot/Tik-110.501/1995/ban.html`.

[80] Charles Landau. CapROS: The capability-based reliable operating system. `http://www.capros.org/`.

[81] Henry M. Levy. *Capability-based Computer Systems*. Digital Press, Bedford, MA, 1984.

[82] Frank Leymann. Web Services Flow Language (WSFL 1.0). `http://www-3.ibm.com/software/solutions/webservices/pdf/WSFL.pdf`.

[83] Ninghui Li, Benjamin N. Grosof, and Joan Feigenbaum. Delegation Logic: A logic-based approach to distributed authorization. *ACM Transactions of Information and System Security (TISSEC)*, 6(1):128–171, Feb 2003.

[84] Tie Liao. Light-weight reliable multicast protocol (LRMP). Technical report, INRIA, Oct 1998.

[85] Sergio Marti and Hector Garcia-Molina. Taxonomy of trust: Categorizing P2P reputation systems. *Computer Networks*, 50(4):472–484, 2006.

[86] Pietro Michiardi and Refik Molva. Core: a collaborative reputation mechanism to enforce node cooperation in mobile ad hoc networks. In *Proceedings of the IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security*, pages 107–121, Deventer, The Netherlands, 2002. Kluwer, B.V.

[87] Mark Miller, Ka-Ping Yee, and Jonathan Shapiro. Capability myths demolished. `http://zesty.ca/capmyths/`.

[88] Robin Milner. *Communicating and Mobile Systems: The Pi-Calculus*. Cambridge University Press, Cambridge, United Kingdom, 1999.

[89] Naftaly H. Minsky and Jerrold Leichter. Law-governed Linda as a coordination model. In *ECOOP '94: Selected papers from the ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution, Object-Based Models and Languages for Concurrent Systems*, pages 125–146. Springer-Verlag, 1995.

[90] Roland Mittermeir and Lydia Würfl. Greedy reuse: Architectural considerations for extending the reusability of components. In *Proceedings of SEKE'96, the Eighth International Conference on Software Engineering and Knowledge Engineering*, pages 434–441, 1996.

[91] Tim Moreton and Andrew Twigg. Enforcing collaboration in peer-to-peer routing services. In *Proceedings of the First International Conference on Trust Management*, pages 255–270, May 2003.

[92] Anthony Nadalin, Chris Kaler, Ronald Monzillo, and Phillip Hallam-Baker. Web services security: SOAP message security 1.1 (WS-Security 1.1). `http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf`.

[93] S. V. Nagaraj. Access control in distributed object systems: Problems with access control lists. In *WETICE '01: Proceedings of the 10th IEEE International Workshops on Enabling Technologies*, pages 163–164, Washington, DC, USA, 2001. IEEE Computer Society.

[94] National Institute of Standards and Technology. Digital Signature Standard, FIPS PUB 186. `http://www.itl.nist.gov/fipspubs/fip186.htm`, May 1994.

[95] National Institute of Standards and Technology. Secure Hash Standard, FIPS PUB 180-2. `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`, Aug 2002.

[96] Dan M. Nessett. A critique of the Burrows, Abadi and Needham logic. *SIGOPS Oper. Syst. Rev.*, 24(2):35–38, 1990.

[97] Peter Newton and James C. Browne. The CODE 2.0 graphical parallel programming language. In *ICS '92: Proceedings of the 6th International Con-*

ference on Supercomputing, pages 167–177, New York, NY, USA, 1992. ACM Press.

[98] OASIS Consortium. The Universal Description, Discovery, and Integration (UDDI) standard (web site). `http://www.uddi.org`.

[99] Karol Ostrovský. *Higher Order Broadcasting Systems*. PhD thesis, Chalmers University of Technology and Göteborg University, 2002.

[100] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford Digital Libraries Project, 1999.

[101] George A. Papadopoulos and Farhad Arbab. Coordination models and languages. Technical Report SEN-R9834, Centrum voor Wiskunde en Informatica (CWI), 1998.

[102] Doron A. Peled. *Software Reliability Methods*. Springer-Verlag, New York, NY, USA, 2001.

[103] Gian Pietro Picco and Marco L. Buschini. Exploiting transiently shared tuple spaces for location transparent code mobility. In *Coordination Models and Languages, Proceedings of COORDINATION 2002 (LNCS 2315)*, pages 258–271. Springer-Verlag, 2002.

[104] Amir Pnueli. The temporal logic of programs. In *Proc. 18th Ann. IEEE Symposium on Foundations of Computer Science*, pages 46–57, 1977.

[105] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Schenker. A scalable content-addressable network. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, New York, NY, USA, 2001. ACM Press.

[106] Paul Resnick, Ko Kuwabara, Richard Zeckhauser, and Eric Friedman. Reputation systems. *Communications of the ACM*, 43(12):45–48, 2000.

[107] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the*

18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), LNCS 2218, pages 329–350. Springer-Verlag, Nov 2001.

[108] Antony Rowstron and Alan Wood. An efficient distributed tuple space implementation for networks of workstations. In Luc Bougé, Pierre Fraigniaud, Anne Mignotte, and Yves Robert, editors, *EuroPar 96 (LNCS 1123)*, pages 511–513. Springer-Verlag, 1996.

[109] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel. SCRIBE: The design of a large-scale event notification infrastructure. In *Networked Group Communication*, pages 30–43, 2001.

[110] Ravi S. Sandhu. Access control: The neglected frontier. In *First Australian Conference on Information Security and Privacy*, 1996.

[111] Ravi S. Sandhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, Feb 1996.

[112] Ravi S. Sandhu and Pierangela Samarati. Access control: Principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[113] Karthikeyan Sankaralingam, Simha Sethumadhavan, and James C. Browne. Distributed pagerank for P2P systems. In *HPDC '03: Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing*, pages 58–68, 2003.

[114] Jonathan Shapiro. What *is* a capability, anyway? `http://www.eros-os.org/essays/capintro.html`.

[115] Jonathan S. Shapiro. *EROS: A Capability System*. PhD thesis, University of Pennsylvania, 1999.

[116] Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[117] Gregg Tally, Daniel Sterne, Durward McDonnell, David Sherman, David Sames, Pierre Pasturel, and John Sebes. A scalable approach to access control in distributed object systems. *NAI Advanced Security Research Journal*, 1(1):75–93, 1998.

[118] Andrew S. Tanenbaum, Sape J. Mullender, and Robbert van Renessee. Using sparse capabilities in a distributed operating system. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 558–563, May 1986.

[119] Mary Thompson, William Johnston, Srilekha Mudumbai, Gary Hoo, Keith Jackson, and Abdelilah Essiari. Certificate-based access control for widely distributed resources. In *Proceedings of the Eighth Usenix Security Symposium*, pages 215–228, Aug 1999.

[120] Robert Tolksdorf and Gregor Rojec-Goldmann. The SPACETUB models and framework. In *Coordination Models and Languages, Proceedings of COORDINATION 2002 (LNCS 2315)*, pages 348–363. Springer-Verlag, 2002.

[121] Mahesh V. Tripunitara and Ninghui Li. Comparing the expressive power of access control models. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 62–71, Oct 2004.

[122] John Turek and Dennis Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, Jun 1992.

[123] United Devices. Grid MP (web site). `http://www.ud.com/products/gridmp.php`.

[124] Amin Vahdat, Eshwar Belani, Paul Eastham, Chad Yoshikawa, Thomas Anderson, David Culler, and Michael Dahlin. WebOS: Operating system services for wide area applications. In *Proceedings of the Seventh Symposium on High Performance Distributed Computing*, 1997.

[125] Jesse Walker. Unsafe at any key size: An analysis of the WEP encapsulation. Technical Report 03628E, IEEE 802.11 Committee, Mar 2000.

[126] Jinghua Wen, Mei Zhang, and Xiang Li. The study on the application of BAN logic in formal analysis of authentication protocols. In *ICEC '05: Proceedings*

of the 7th international conference on Electronic commerce, pages 744–747, New York, NY, USA, 2005. ACM Press.

[127] Wi-Fi Alliance. Wi-Fi Protected Access: Strong, standards-based, interoperable security for today's Wi-Fi networks (white paper). `http://www.wifialliance.com/OpenSection/pdf/Whitepaper_Wi-Fi_Security4-29-03.pdf`.

[128] Shiow-Yang Wu, Daniel P. Miranker, and James C. Browne. Decomposition abstraction in parallel rule languages. *IEEE Transactions on Parallel and Distributed Systems*, 7(11):1164–1184, Nov 1996.

[129] WWW Consortium. Web serices activity (web site). `http://www.w3.org/2002/ws`.

[130] Xiaowei Yang, David Wetherall, and Thomas Anderson. A DoS-limiting network architecture. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 241–252, New York, NY, USA, 2005. ACM Press.

[131] Po-Wah Yau and Chris J. Mitchell. Reputation methods for routing security for mobile ad hoc networks. In *Proceedings of SympoTIC '03*, pages 130–137, Oct 2003.

[132] Ben Y. Zhao, Ling Huang, Jeremy Stribling, Sean C. Rhea, Anthony D. Joseph, and John Kubiatowicz. Tapestry: A resilient global-scale overlay for service development. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan 2004.

Vita

Kevin Michael Kane was born in Washington, D.C. on July 24, 1978, the son of Ronald Lawrence Kane and Carol Mary Kane. After completing his work at Rockville High School, Rockville, Maryland, in 1996, he entered the University of Maryland in College Park, Maryland. He received the degree of Bachelor of Science with Honors in Computer Science from the University of Maryland in May 2000. In August 2000 he entered the Graduate School of The University of Texas at Austin. He received the degree of Master of Science in Computer Sciences from the University of Texas at Austin in December 2005.

Permanent Address: 9511 Quail Village Lane
Austin, TX 78758-5815

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1] LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.