Negotiating Agreements Using Policies in Ubiquitous Computing Scenarios

V. Ramakrishna
Computer Science Department, UCLA
vrama@cs.ucla.edu

Kevin Eustice
Computer Science Department, UCLA
kfe@cs.ucla.edu

Peter Reiher
Computer Science Department, UCLA
reiher.cs.ucla.edu

Abstract

The emerging ubiquitous computing vision is characterized by decentralized and ad hoc interoperation among devices and networks for access to services. Interacting devices or groups have highly heterogeneous resources and security and privacy concerns, and invariably belong to different security or administrative domains. Flexible and automated mechanisms are needed to achieve effective cross-domain interoperation that leads to a service or resource sharing agreement. We describe how policies representing system state, requirements and intent can be used to negotiate agreements between mutually unknown and untrusted systems that differ widely in their characteristics. Our negotiation protocol uses a small number of message types, which we have found to be sufficient for supporting a wide variety of application scenarios that occur on the Web, and that will likely be important in the ubiquitous computing environments of the future.

1. Introduction

The ubiquitous computing (*ubicomp*) vision of access to computing and network services everywhere and at any time is gradually being realized. *De facto* standards such as TCP/IP and 802.11 based MAC protocols enable mobile devices to connect and obtain services for their users. However, spontaneous ad hoc *interoperation* among systems (typically mobile devices and host networks) within different administrative and security domains is far from a reality. Interoperation here refers to the ability to communicate information about objects, offer services, discover external services, and access external resources. Ensuring suitable configuration and compatibility is hard, given the heterogeneity of resource capabilities, services offered, and credentials possessed by the interacting devices and networks. Interacting domains have different security, privacy and resource constraints, which vary with changing context. Ensuring proper static configuration for seamless service access in all possible contexts is impractical [22], and a centralized solution is neither scalable nor desirable. Ubicomp interoperation should also be highly automated, requiring minimal decision-making by its users.

Existing solutions for cross-domain interoperation fall into two extremes: i) freely discoverable and accessible services, which are vulnerable to abuse, or ii) tight and inflexible security models that require prearranged trust relationships. A more flexible solution would balance the needs of service access and security. Interactions also involve give-and-take by both parties, each acting as a service producer and a consumer. To maintain privacy and prevent resource abuse, it is undesirable to expose local rules and constraints to external parties without having some level of trust.

Consider a motivating scenario, where interoperation fails without prior configuration due to stringent security policies and a lack of preestablished trust. An ACM-conducted conference is being held in a room containing a wireless network, a display device, a projector and a printer. To access conference room services, an attendee would have to manually configure his device. Many attendees have ACM credentials (e.g., certificates), which should ideally permit automated configuration. All that is needed is an effective procedure for the network to ask for proof of satisfactory accreditation, verify it and grant access; the user's device is then ready to provide private information in return for needed services. The network could impose policy constraints on the requesting device, such as an audio-silent request during the conference, and grant service access only upon compliance. The conference system might choose to offer further privileges (such as copies of presentations or papers) to users who subscribe to a journal related to the conference. These tasks can be achieved through a step-by-step progression of trading information and making agreement decisions.

The dynamics of this scenario are similar to how web services are accessed today. Clients and servers interact through rigid protocols, often with user input, because of the lack of a viable procedure for dynamic agreement, projects like P3P [17] notwithstanding.

We propose a solution for interoperation between two computers through a simple, generic, and automated *negotiation* protocol, applicable to a wide range of ubiquitous computing scenarios and services. The common basis or abstraction of negotiation is the set of *policies* (description of system state, constraints and desired behavior) that every interacting entity must possess. Given that the negotiation parties possess policies specified in a common logic-based semantic language, we show how they can come to a mutually satisfactory resource-sharing and service-access agreement. No common prior configuration or trust relationship is required. The minimum requirement is a shared high level ontology and language that allows a common understanding of resource types and properties. Negotiation is a multi-step exchange of requests, counter-requests, and offers, guided by policy at every step. Since the system policy is the only variable, the protocol is flexible and responds dynamically to context changes. Using policies to control system behavior, enforce security constraints, and attain goals is not a new idea. Enabling dynamic enforcement of such policies for interoperation is our contribution; we treat expressivity as a secondary concern.

In Section 2 we describe how policies are specified at the negotiating end points. We describe the process and semantics of the negotiation framework in Section 3. Our implementation of this framework using the *Panoply* ubiquitous computing middleware [8] and the ensuing experiments are described in Section 4. We conclude with discussions of applications and extensions.

2. Policy Language Features and Semantics

Our negotiation framework depends on the interacting entities (single computing devices, networks, or clusters of devices) possessing policies that describe their state and constraints. Policies are sets of rules that describe the behavioral constraints and ideals of a computing system. Examples include policies for resource management, security and access control, and context adaptation. Developing a policy language was an important part of our research, though not its focus. We had to identify a small set of requirements for a policy language that would be independent of the domain or applications it was used in, and still be usable by the negotiation framework in diverse scenarios. The constructs or syntactic features provided by the language were important only to the extent that scenarios could be constructed. To minimize our initial effort, we chose to adapt a policy language from an existing language that would provide a syntactic base and basic reasoning semantics, retaining the option of modifying or augmenting it later. Though a number of desirable features of a policy language can be found in existing languages (see Section 5), the combination of features needed for a negotiation framework was not available in any existing policy language.

We determined that our policy language had to be a based on a formal logic and have well-defined reasoning semantics, as: i) this makes the policy language usable in domains with diverse resource capabilities and constraints, without being tied closely to any one in particular [12]; and ii) decision-making on the basis of policies described in a logic-based language ensures correct and consistent behavior. The language should allow easy specification of intent and goals, leaving the enforcement procedure to the runtime system. Thus, while the vocabulary describing objects could be specific to a domain, the semantics of dealing with the specifications will be common across domains. Most system information must be understood only within a limited domain, and need not be part of a global specification language. Interoperating devices still must understand and interpret the objects that they trade. Such specification issues have been researched in the context of the Semantic Web and other open frameworks, examples being RDF/XML (which have been widely adopted), DAML+OIL [5] and OWL [16]. Our policy ontology is inspired by SOUPA [1], which defines a set of *core* components and optional *extensions* that can be used to model ubiquitous computing applications. Our ontology is fairly informal at this stage, but it includes the following: *entities* and *agents*, *resources* and *content*, *properties* and *metadata*, *mechanisms* (e.g., *sensory*, *networking*, *cryptography*), *context*, *relationships* between entities and resources, *quantitative limits*, *precedence rules*, *deontic constraints* (e.g., *permission*, *obligation*), *actions* and *events*.

Our policy language is built on Prolog, which is based on first-order logic. Recent implementations [18] have significantly advanced its computational efficiency, making Prolog satisfactory for use in a real-world framework, especially as mobile users are unlikely to perceive appreciable response time lag (see Section 4.4). The structure of predicates, variables and other terms in Prolog allows us to specify categories and instances of entities, objects and contextual parameters in policy rules. The semantic nature of a logic-based policy language also enables specification of high and low level policies, and the specification of relations between these. We use the SWI-Prolog code base and API [21], which offer important features that will be discussed later. System state and policy rules are defined in the form of Prolog facts and rules (clauses). Examples are given below. Clauses 4 and 5 illustrate *if-then* policies.

1) *fileType('song.mp3',audio).* **['song.mp3' is an audio file]**
2) *relation(alice,bob).* **['alice' and 'bob' are relations]**
3) *certificate('UCLA').possess(john,'UCLA').* **['UCLA' is a certificate, and is possessed by 'john']**
4) *member(X) :- candidate(X), teamMember(X), numChildren(N), maxChildren(M), N<M.* **[X is a member if it is a candidate, and a team member, and if the number of current children is less than the maximum]**
5) *access(S,V) :- candidate(S), teamMember(S), voucher(location,V).* **[entity S can be granted access to voucher V if S is a 'candidate' and a team member, and if V is a 'location' voucher]**

Since Prolog does not completely obey first-order logic semantics, we restricted the syntactic features of our language to make the reasoning algorithms sound. The syntax consists of standard conjunction, disjunction, and implication operators. Function definitions are not allowed; they are specified as relations, or predicates with true/false values, thereby avoiding the *occur-check* [14] problem. Cyclic predicate definitions are not allowed, guaranteeing that query processing through the backward chaining algorithm terminates. We use Prolog's *negation by failure* feature to prove negatives.

Policies are managed in a single database of Prolog facts and rules, which can be accessed and manipulated through meta-predicates defined by SWI-Prolog. We use higher-level predicates, e.g., *assert, retract, clause*, and *functor*, to manipulate and examine the database and individual rules. Rule indexing and retrieval impact system performance, and the core mechanisms provided by SWI-Prolog proved reasonably efficient. SWI-Prolog also provides a bidirectional Java to Prolog API, which is valuable since the negotiation framework is implemented in Java. The following examples illustrate low level rules: in (6), a low-level JPL query (jpl_call) is required to call a Java method to translate a group member name to a player name; and in (7), the action of closing a port 'Po' requires the execution of the 'iptables' shell command.

```
6) teamMember(X) :- groupMember(X), playerName(Y),
      jpl_call('panoply.policy.Helper','sphereName',[X],Y).
7) action(closePort,Po) :- atom_concat('iptables -A INPUT -j
      DROP -p tcp --dport ',Po,C1), atom_concat(C1,' -i lo',C),
      shell(C,0).
```

We have added mechanisms for the addition, removal and modification of state information and policy rules in a database. Methods for examination of policy rules, used for negotiation message generation, are described in Section 3. We define a special category of *event-action-trigger* rules that respond to events. An example of such *update* rules is given below:

```
8) update :- (((numRelatives(X,N), door(X), closeD(X)),
      doorOpen(X), N>0) →
      (retract(doorOpen(X)), retract(closeD(X)))).
```

When a Prolog query to 'update' is made, all such update rules are evaluated; if all conditions on the antecedent of the body of the clause are proved true, the consequents are also evaluated. In (8), 'update' results in a change to the state of a ubiquitous door service. Such policies are examined whenever an addition or removal is made to the database, the result being that appropriate state changes are made in response to policy-specified events.

3. Negotiation Framework

In this section we describe our framework for automated negotiation among ubiquitous computing entities. We focus on two-party negotiation, but our model can potentially be extended to multi-party negotiation.

We have discussed how policies can be used to describe state information and system constraints. In our ubiquitous computing model, interacting entities possess services, resources and policy rules, all of which may be unknown to the other. Each entity also has goals that can be achieved only through interoperation; in a large class of scenarios, only one party (e.g., a mobile device discovering and joining a network) starts off with goals.

3.1. Negotiation Model

Negotiation is a policy-guided operation by which devices request and grant services from each other. Each participant's local policies are private and unknown to the other, and they might conflict. Keeping policies private is practical, since exposure of certain policies could open a system to abuse. For example, the knowledge that resource R can be accessed only between 8 pm and 9 pm, and only by X, could invite denial-of-service attacks targeted both at the resource host and at X. Each entity starts with certain requirements, or targets, or goals; negotiation guides entities to agreement or *compromise* through the use of meta-policies, heuristics and logical reasoning. It is, in effect, a decentralized process of policy resolution and conflict management, except that each entity has partial knowledge of the other's policy, state and goals. Negotiation is bi-directional, neither party being a client or a server, since both entities may possess objects that the other desires. For example, a patron's PDA and a coffee shop network could derive mutual benefit from interaction; the former obtains network access, while the latter could expand its customer base through incentives that include network access.

In our negotiation model, we assume that each party already knows about the other's presence and has a low-level data communication channel with it. Negotiation starts with both entities attempting to attain their goals, i.e., obtaining services from the other through queries and responses. At every negotiation step, each party discovers more characteristics, services and policy constraints possessed by the other party, and this discovery triggers a reevaluation of their policies and their original goals.

We describe negotiation here in semi-formal terms. Negotiators C_1 and C_2 possess sets of resources and services S_1 and S_2, policy sets P_1 and P_2, and goal or requirement sets G_1 and G_2, respectively. Neither party has any knowledge of the other's sets. At the end of negotiation, C_1 will have access to a set of services $Q_1 \subseteq S_2$, and C_2 access to a set of services $Q_2 \subseteq S_1$. This assignment will be consistent with both entities' policy sets and will satisfy part of the requirements of the respective goal sets. An ideal negotiation would result in the maximal possible sets for Q_1 and Q_2, but this may not possible without an oracle that has complete knowledge of both parties. The negotiation protocol that we have designed and implemented is a *best-effort* solution, and does not provide theoretical guarantees of optimality in terms of the agreement reached.

3.2 Protocol Units and Semantics

Our negotiation protocol is a lightweight message-exchanging procedure with a small number of message types that we determined were necessary and sufficient for most negotiation scenarios, the contents of which can be interpreted in a domain-independent manner. This is necessary for both flexibility and extensibility. Therefore, the negotiation messages in our protocol design are based on *illocutionary speech acts* [19], which are simple generic utterances of intent to perform particular actions or to make wishes known; for example, assertion, suggestion, promise, command, etc. Our negotiation protocol has a few high-level message types:

- *requests*: access rights, permission to perform actions, information queries.
- *offers*: replies to request(s), with supporting objects, proofs, information; e.g., data files, certificates, etc.
- *policies*: rules or obligation statements that the opposite entity must comply with.

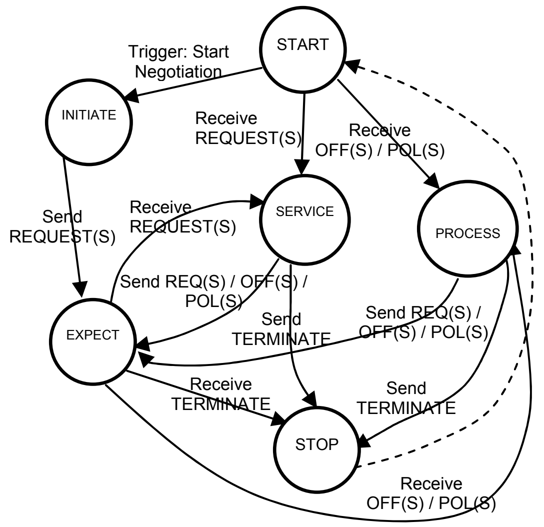

Figure 1. Negotiation Protocol State Machine

Both negotiators run the state machine in Figure 1 and send messages based on these high-level speech acts. Other kinds of illocutionary speech acts, such as commands, affirmations, declinations and promises, are subsumed within the above set and do not require special handling at the state machine level.

A negotiation is triggered by one of the entities but then proceeds in a peer-to-peer fashion. Negotiation starts with one entity making request(s) of the other; these requests are derived from the initiator's goals and requirements encoded in policy. Offers or counter-requests could be made in response. Every negotiation message is identified by its high-level type, and consists of a set of policy-derived statements. A request message contains a set of predicates drawn from the common vocabulary and indicates certain wishes on the sender's part. Request message entries are drawn from a small set, including *possessions* (e.g., a wish to obtain possession of an object, or access to a service owned by the opposite party), *actions* (e.g., permission to perform certain actions, such as running an application, or commanding the other to run a piece of code), *state changes* (e.g., permission to join the host network), or a simple *query* (e.g., asking about the resolution of a color display possessed by the other). Correspondingly, offers would contain either an acceptance or a rejection of the request. For acceptances, the object of the request is appended to the message (for example, proof of possession of a credential in the form of a certificate, or an answer to a query as a string). Every negotiation protocol message contains multiple entries for efficient communication.

Counter-requests can be generated in response to a received request, resulting in a request queue on either side. A request is dequeued when an offer is accepted or rejected. The protocol terminates when either side has both its sent request and received request queues empty.

The state machine is non-deterministic and presents various choices at every state. Local policy and message contents determine the choices made. We kept the state machine simple to make it generic and capable of supporting a wide variety of applications.

3.3. Policy-Guided Reasoning Mechanisms

The negotiation framework parses request and response messages, determining appropriate actions by examining and manipulating the policy database.

Request Processing: When a negotiator receives a request, every entry is extracted and queued. A query is run on every entry to test whether that request can be granted based on the current policy set. If it can, an affirmative offer is noted; eventually, offers are sent in a batch for optimized communication. If the request cannot be granted, counter-requests are generated through the following procedure. First, the sets of policies that govern the request are selected. For example, if a request for access to a service is made, policies of the form

$$access(ServiceName) \text{ :- } pred_1 \,\&\, pred_2 \,\&\ldots\ldots$$

are selected and parsed. Our policies are conjunctions of conditions. The overall policy governing access to the service is a *disjunction* of the *sets of conjunctions* that make up the bodies of these policy statements. Conjuncts in each such policy statement are evaluated; if the result is *true*, and the predicates represent objects or actions that can be requested, they are added to a queue. If an unsatisfied predicate in the body is not part of the shared global vocabulary, the algorithm recursively examines all policies having that predicate as the head until *leaves* (or facts) are reached. For example, a policy of the following structure would lead to recursion:

$pred_2$:- $pred_{21}$ & $pred_{22}$ &……

At the end of this procedure, multiple sets of request predicates are generated. Each set is a collection of counter-requests to be returned to the other party; if the opposite party satisfies this set of requests, its original request will be granted. If multiple sets are generated, one is immediately sent, and the rest saved as alternatives in case the opposite party is unable to satisfy the ones sent earlier. If no counter-requests can be generated, or no alternatives remain, a rejection offer is sent.

Offer Processing: If an acceptance offer is received, the request that was satisfied is popped from the queue. If requests A, B, and C were sent as counter-requests in response to a request R received, and if acceptance offers were received for all three, A, B, and C are popped from the *sent requests* queue, and R is removed from the *received requests* queue. On the other hand, if a rejection is sent, an alternative set of counter-requests (computed during request processing) is sent back. In some cases, an acceptance offer is received that does not contain valid supporting objects; e.g., a certificate does not validate, or a request for closing port 25 was not met. In this case, an offer rejection message is sent. The other party could re-send a correct offer, an alternative offer, or a rejection.

Negotiation messages containing policy statements demand compliance with particular policy rules, so they are handled like requests. We separate policies and requests for flexibility: instead of specific requests, an entity can send its policy and let the opposite party determine a means of compliance. Also, different "secrets" are revealed by a request and a policy, and we want to control what information is released when. Though our current applications of negotiation mainly involve requests and offers, research in enhancing the negotiation protocol with rich strategic control is ongoing.

4. System Implementation

To demonstrate how our negotiation system would interface with a full-featured ubicomp platform, we describe how negotiation and policy management works in Panoply [8]. Panoply provides functions for applications to manage individual devices and groups. The policy management and negotiation mechanisms are independent of the Panoply design, and could work for other ubicomp platforms with minor adjustments.

4.1. Spheres of Influence and Panoply

The core representational unit of Panoply is the *Sphere of Influence*, which can represent an individual device or a group of devices united by a common interest or attribute such as physical location, application, or social relationship. Spheres unify disparate notions of "groups", such as device clusters and social networks, by providing a common interface and a standard set of discovery and management primitives. Spheres map onto the *domains* discussed in Section 1.

Panoply provides group management primitives that allow the creation and maintenance of spheres of influence, including discovery, joining, and cluster management. Intra- and inter-sphere communication is via a publish/subscribe event model that propagates events between devices and applications, subject to scoping constraints embedded in events and interest. There is a symbiotic relationship between the Sphere of Influence framework and the policy-guided negotiation framework. The latter ensures the integrity of sphere joining and communication, and provides a security blanket. Panoply ensures that appropriate events are sent to the policy manager, enabling it to keep its database updated and make suitable decisions about when and how to negotiate.

4.2. Policy Management within Panoply

The policy manager is a module of the Panoply middleware that serves as a container for system (or sphere) policies which are enforced through both passive and proactive means. The architecture can be functionally decomposed into three layers (see Figure 2).

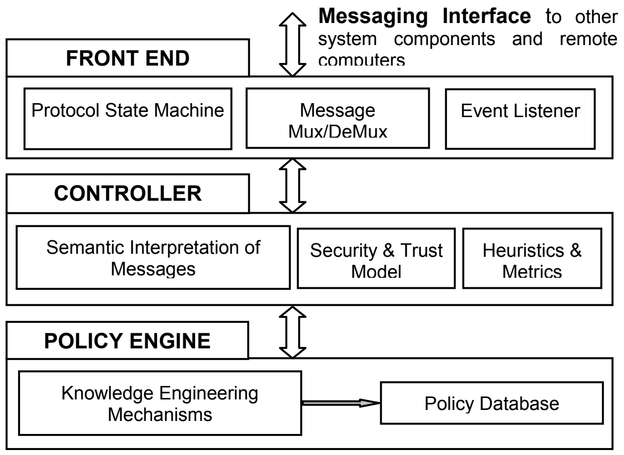

Figure 2. Policy Manager Functional Diagram

The *front end* is the policy manager shell that interfaces with other local sphere components, as well as interacting with remote spheres through Panoply events. It receives state change events from the sphere and applications, and communicates them to the policy engine, which updates the policy database and triggers suitable actions. It mediates information flow to Panoply applications by monitoring events. The negotiation protocol state machine runs here. Multiple simultaneous negotiation threads, and the flow of negotiation messages, are managed at this layer. Concurrent independent negotiations with multiple peers are supported, though all threads share a common policy database. The front end allows users to observe and modify policies with a graphical interface.

The *policy engine* manages the database containing state information and policy rules. It interfaces with the SWI-Prolog database and exports methods for manipulating the database and extracting information. These methods include the simple querying operations, addition and removal operations, and the policy examination and the request-generation mechanisms.

The *controller* guides and controls the rate and the strategy of negotiation. Every negotiation thread has its own controller. It examines negotiation message contents, the request queues, and sets of alternative requests and offers. Individual message entries are extracted and analyzed based on protocol semantics. Such analysis includes inferring why those entries were sent, how to process them, and what messages to send in response. As described in section 3.2, negotiation messages contain queries, responses, and supporting objects ranging from string tokens to credentials, data files and mobile code. Part of the controller's task is to extract or attach suitable objects. For example, in the case of a message containing a *request* to run a virus-scanner, the controller would find the scanning code and attach it to the message. In a message containing an *offer* of a certificate, the controller would look for, extract, and validate the certificate object; different decisions are made depending on whether the correct offer was sent. How a message is processed depends on the message type and the objects it carries. We cannot handle all possible objects of interest in every ubicomp scenario, so the controller allows pluggable helper functions and mechanisms for different objects as the need may arise. The controller interfaces directly with the policy engine and uses the methods exported by it to drive the negotiation strategy. Our controller framework, though powerful, is fairly basic in its design. We are investigating a more dynamic negotiation control that sends messages based on game-theoretic strategies. Heuristics will be based on incentives and risks, backed by trust and utility models [7].

4.3. Example Negotiation Scenarios

We experimented with multiple negotiation scenarios within the Panoply framework. Since the unit of interaction in the Panoply model is a sphere, the examples involved two spheres negotiating to gain certain privileges, chiefly the ability to become a member of another sphere. For example, a host sphere has the following policy for granting membership to a supplicant:

```
member(X) :- candidateSphere(X), action(X,order,run,'uname
   -a | cut -f 3 -d \' \'',JVerInfo), JVerInfo='2.6.17.1',
   closedPort(X,25),
   possess(X,U),socialVoucher(U,G,G),localSphereID(G).
```

This indicates that membership can be granted only if the supplicant is a candidate sphere, runs a Linux kernel version 2.6.17.1, has network port 25 closed, and proves possession of a voucher credential granted by the host. The negotiation starts by the supplicant requesting membership in the sphere. The host generates a request message containing a query for the OS kernel version, a request to shut port 25, and proof of possession of a "social" voucher. The supplicant's policies allow the release of kernel version information and the shutting of certain network ports, but it will not release its private credential without the host providing a valid UCLA voucher. (We do not list the policies here due to limited space). It sends a counter-request for proof of a UCLA credential, which the host releases. The supplicant then satisfies all the original requests. Finally the host returns a positive response to the membership request, and makes appropriate configuration changes. The supplicant, having been satisfied, terminates the negotiation session.

This is a simple example. The negotiation takes only a few steps, and few policy rules are involved. But our model can handle arbitrarily complex policies and requests for a wide variety of services. We use negotiation for membership in various real sphere applications and for access control in Panoply through event flow mediation.

For example, we built a smart party application for dynamically choosing songs to play, and a GUI for manually changing their order. Party host spheres are assigned credentials in the form of vouchers. The sphere running the song-playing application will only allow designated party hosts to control the GUI through the following policy: "*A control event destined for the song-playing application is allowed to pass only if the event sender is a valid party host.*" Every control event is redirected to the policy manager upon arrival. If it cannot establish that the sender was a valid host, it triggers a negotiation requesting proof of possession of a party host voucher. If a satisfactory offer is received, negotiation terminates and the event is passed to the application; otherwise the event is dropped.

4.4. Evaluation

Automated negotiation is a fresh approach for dynamic service discovery and access in ubicomp environments. As our experience with building and testing a policy manager in Panoply shows, we created a negotiation framework independent of particular devices, domains and applications. Our negotiation procedure ensures consistency with local policy, since no decisions are made during negotiation that would conflict with a policy rule. The protocol is guaranteed to terminate in a finite number of steps, because the policy database is of finite size, and every policy statement is of finite length. Though not completely tolerant to network failures, our framework could directly make use of existing research.

A detailed performance analysis is beyond the scope of this paper, but we show sample timings for certain characteristic negotiations in Table 1. The negotiation was conducted between two spheres, N1 running on an IBM Thinkpad T42 (1.7 GHz, 512 MB) laptop and N2 running on an Intel P4 (2.53 GHz, 512MB) desktop; both machines ran Linux. Messages were exchanged through a TLS connection running over an 802.11b wireless channel. The table indicates the times taken to complete negotiation, from the first message sent or received until termination. P indicates the local processing time, W the wait time when expecting a reply from the negotiator, and 'Total' is the total time (Total=P+W). In each case, Negotiator 1 initiates the protocol by sending a request message. R(*k*) indicates that *k* is sending a request to its negotiator, AO(*k*) indicates an affirmative offer sent by *k*, NO(*k*) a negative offer, and T(*k*) indicates termination. '3 C-R(*k*)' indicates three counter-requests sent by *k*.

Table 1. Sample Negotiation Performance Measurements (in *milliseconds*)

Case I: R(N1) → AO(N2) → T(N1)
Case II: R(N1) → NO(N2) → T(N1)
Case III: R(N1) → 3 C-R(N2) → 3 AO(N1) → AO(N2) → T(N1)
Case IV: R(N1) → C-R(N2) → NO(N1) → 3 C-R(N2) (alternative) → 3 AO(N1) → 1 AO(N2) → T(N1)

	Negotiator 1 (N1) Timing			**Negotiator 2 (N2) Timing**		
	P	W	Total	P	W	Total
I	180.5	1081.2	1261.7	97.5	1958.8	**2056.3**
II	8.7	1124.6	1133.3	112	1897.8	**2009.8**
III	387.8	4251.2	4639	2231.1	3167.8	**5398.9**
IV	504	6871.5	7375.5	3178.7	4886.5	**8065.2**

All numbers are reported with 99% confidence intervals whose widths are typically ~5% of their mean. 'Tot' includes the message processing overhead introduced by the Panoply middleware, which explains the discrepancy between the processing and total times. Entries in boldface indicate which negotiator's time dominates the other. As we can see, simple negotiations take a few seconds (~8) to terminate, and additional steps introduce a small linear overhead. Our counter-request generation algorithm introduced reasonable overheads, at N2, of 1710.5 msec in Case III and 2573.8 msec in Case IV. Using external methods to verify vouchers took ~15 msec, which is small, but running shell commands and executing code may introduce larger overhead. In practice, a few seconds overhead for negotiation will not be noticed by users in a majority of ubicomp scenarios. Ongoing research will reduce the costs of negotiation.

5. Related Work

The problem of interoperation for resource access is widely recognized as important, and existing technologies are inadequate for dealing with strangers and dynamic context changes [22]. Many research efforts have investigated the use of policies to specify goals, and control systems and security, because of their flexibility and adaptability [20]. Such research has primarily focused on policy specification and expressivity as compared to the interoperation mechanism, but we have borrowed concepts from several policy languages. The most relevant policy language is Rei [12], which is targeted towards both ubicomp and the Semantic Web. Rei is based on domain-independent logical semantics, and adds support for specification of actions, speech acts, and modality to other standard features. Other languages either have restrictive semantics, like Ponder [4], or have restricted application models, like IBM's Trust Policy Language [11]. Semantic Web technologies like RDF/XML, DAML+OIL [5], OWL [16], and SOUPA [1] have contributed towards cross-domain communication, and are largely complementary to our work.

Our work advances research in universal spontaneous interoperation for services by enabling dynamic agreements based on variable policy, going beyond frameworks like Jini [24] and UPnP [23], which are inflexible in some aspects, especially security.

Our negotiation framework can also be viewed as a process of gradually building up trust and controlling access to services. Advanced role-based access control systems, such as GRBAC [2] and dRBAC [9], and law-governed interaction [15], provide more expressive and flexible policy-based access control than traditional ACLs and capabilities, but do not handle conflicts and disagreements, where negotiation would be required.

The most related work is *automated trust negotiation* [25], through which web entities (typically client-server, but also peer communications) can establish trust to obtain access to a guarded resource, resulting in a grant or denial of access. The negotiation protocol involves progressive request and exposure of sensitive credentials evaluated using per-credential access control policy rules. The TrustBuilder system [25], based on TLS, implements trust negotiation as a more flexible policy-guided access control mechanism. It suffers some drawbacks for ubicomp interoperation. The policy language is not based on logical semantics, and requests can be made exclusively for credentials. Conflicts cannot be resolved through compromises or alternatives. PeerTrust [10] uses a distributed logic programming language based on Prolog to advance TrustBuilder concepts and applies a more powerful model for trust negotiation in the Semantic Web. Both PeerTrust and TrustBuilder lack support for context-awareness; their fixed goals cannot be reevaluated during negotiation, and they rely excessively on credential-based trust. Our protocol supports a wider range of speech acts.

Negotiation protocols have been proposed for access to grid services to match resource owner and consumer preferences [13]. SNAP [3] is a service-level agreement protocol for grid resource allocation. These protocols, though effective in their targeted environments, are inflexible and lack support for security and context-awareness. Dang and Huhns [6] propose a protocol to manage multiple concurrent negotiations for services among service providers and consumers. Their negotiation agreements are based on utility functions that must be reconciled, whereas our framework is based on security and resource usage policies.

6. Conclusion and Future Work

Ubiquitous computing is characterized by computing and communication services available almost everywhere. The heterogeneity in the nature of devices and networks that interact, and the diversity of security constraints and resource capabilities that they possess, pose obstacles for spontaneous service discovery and access, especially when there is no prior trust basis among the interacting entities. We have designed and implemented a flexible automated negotiation protocol that allows entities to come to agreement through the use of local policies.

We plan several improvements to our negotiation framework, such as more flexible heuristics and strategies to control the flow of the protocol, and using compromises and security risks, based on perceived benefits and costs. We are also building new applications for negotiation, primarily geared towards ubicomp security. We will also consider integrating technologies like RDF/XML into our policy framework.

7. References

[1] H. Chen, F. Perich, T. W. Finin, and Anupam Joshi, "SOUPA: Standard Ontology for Ubiquitous and Pervasive Applications," *MobiQuitous 2004*: pp. 258-267.

[2] M. J. Covington, M. J. Moyer, and M. Ahamad, "Generalized Role-Based Access Control for Securing Future Applications," *23rd National Information Systems Security Conference*, Baltimore, MD, Oct. 2000.

[3] K. Czajkowski, I. Foster, C. Kesselman, V. Sander, and S. Tuecke, "SNAP: A Protocol for Negotiating Service Level Agreements and Coordinating Resource Management in Distributed Systems," *8th Workshop on Job Scheduling Strategies for Parallel Processing*, Jul. 2002.

[4] N. Damianou, N. Dulay, E. Lupu and M. Sloman, "The Ponder Policy Specification Language," *Policy Workshop 2001*, Jan. 2001, Bristol, U.K.

[5] "The DARPA Agent Markup Language Homepage," http://www.daml.org.

[6] J. Dang and M. N. Huhns, "Concurrent Multiple-Issue Negotiation for Internet-Based Services," *IEEE Internet Computing*, Nov./Dec. 2006, Vol. 10, No. 6, pp. 42-49.

[7] C. English, S. Terzis, and P. Nixon, "Towards Self-Protecting Ubiquitous Systems: Monitoring Trust-based Interactions," *Personal and Ubiquitous Computing Journal*, Vol. 10, Issue 1, Dec. 2005, Springer London, pp. 50–54.

[8] K. Eustice, L. Kleinrock, S. Markstrum, G. Popek, V. Ramakrishna, and P. Reiher, "Enabling Secure Ubiquitous Interactions," *1st Intl. Workshop on Middleware for Pervasive and Ad-Hoc Computing (at Middleware 2003)*, 17 Jun. 2003, Rio de Janeiro, Brazil.

[9] E. Freudenthal, T. Pesin, L. Port, E. Keenan, and V. Karamcheti, "dRBAC: Distributed Role-Based Access Control for Dynamic Coalition Environments," *22nd Intl. Conference on Distributed Computing Systems (ICDCS'02), IEEE Computer Society*, Jul. 2002.

[10] R. Gavriloaie, W. Nejdl, D. Olmedilla, K. Seamons, and M. Winslett, "No Registration Needed: How to Use Declarative Policies and Negotiation to Access Sensitive Resources on the Semantic Web," In Proc. *1st First European Semantic Web Symposium*, Heraklion, Greece, May 2004.

[11] A. Herzberg, Y. Mass, L. Mihaeli, D. Naor, and Y. Ravid, "Access Control Meets Public Key Infrastructure, Or: Assigning Roles to Strangers," *Symposium on Security and Privacy*, pp. 2–14, 2000.

[12] L. Kagal, T. Finin, and A. Joshi, "A Policy Language for a Pervasive Computing Environment," *IEEE 4th Intl. Workshop on Policies for Distributed Systems and Networks*, 2003.

[13] R. Lawley, K. Decker, M. Luck, T. R. Payne, and L. Moreau, "Automated Negotiation for Grid Notification Services," *Euro-Par 2003,* pp. 384-393.

[14] K. Marriott and H. Sondergaard, "On Prolog and the Occur Check Problem," *ACM SIGPLAN Notices*, Vol. 25, Issue 5, May 1989, pp. 76-82.

[15] N. H. Minsky and V. Ungureanu, "Law-governed Interaction: A Coordination and Control Mechanism for Heterogeneous Distributed Systems," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 9, No. 3, pp. 273-305, Jul. 2000.

[16] "OWL Web Ontology Language Overview," http://www.w3.org/TR/owl-features/.

[17] "P3P Public Overview": http://www.w3.org/P3P/.

[18] P. Van Roy, "Can Logic Programming Execute as Fast as Imperative Programming?," *PhD thesis, University of California at Berkeley*, November 1990.

[19] J. R. Searle and D. Vanderveken, "Foundations of Illocutionary Logic," *Cambridge University Press*, Cambridge, UK, 1984.

[20] M. Sloman and E. Lupu, "Security and Management Policy Specification," *IEEE Network, Special Issue on Policy-Based Networking,* (invited) 16(2), Mar. 2002.

[21] "SWI-Prolog's Home," http://www.swi-prolog.org.

[22] A. Toninelli, R. Montanari, L. Kagal, and O. Lassila, "A Semantic Context-Aware Access Control Framework for Secure Collaborations in Pervasive Computing Environments," *5th Intl. Semantic Web Conference*, Athens, GA, Nov. 5-9, 2006.

[23] "UPnP Forum," http://www.upnp.org.

[24] J. Waldo, "The Jini Architecture for Network-Centric Computing," *Communications of the ACM,* Vol. 42, No. 7, pp.76-82, 1999.

[25] M. Winslett, T. Yu, K. E. Seamons, A. Hess, J. Jacobson, R. Jarvis, B. Smith, and L. Yu, "Negotiating Trust on the Web," *IEEE Internet Computing*, Nov./Dec. 2002.