A Hierarchical Approach to Service Negotiation

Glenn Bruns
Bell Laboratories, Alcatel-Lucent
Naperville, IL, USA
Glenn.Bruns@alcatel-lucent.com

Mauricio Cortes
Bell Laboratories, Alcatel-Lucent
Murray Hill, NJ, USA
Mauricio.Cortes@alcatel-lucent.com

Abstract—Service negotiation allows a service client to negotiate with a service provider on the terms of service. Much existing work on service negotiation assumes that a provider will define its negotiation strategy in terms of the state of its resources. This approach can lead to complex strategy and assumes, unrealistically, that providers have full knowledge and control of their resources. We propose a hierarchical model of service negotiation in which negotiation strategy is defined in terms of sub-negotiations with internal or external agents. This model helps to manage the complexity of negotiation strategy by allowing it to be decomposed, with each component having well-defined scope. In this paper we present our hierarchical negotiation model, and a negotiation protocol and negotiation policy language based on it.

Keywords—web services, negotiation, policy, negotiation protocol, service agreements

I. INTRODUCTION

The field of service negotiation seeks to enrich a web services ecology by allowing service clients to negotiate on the terms of a service. Much of the work in the field has focused on negotiation protocols (e.g., [1]), defining such things as the structure of the negotiation process, and how offers are encoded in protocol messages.

Another key element of support for service negotiation is negotiation policy: a means for a service provider to express its negotiation strategy. Some work on service negotiation policy is based on classical models of negotiation, where the goal is to get close to a Pareto-optimal agreement, in which one party can only do better at the other’s expense. In this work (e.g., [2]), negotiation strategy is based on an abstract model of a service provider’s resources. A problem in applying this work is that many providers have resources that are complex and only partly visible to the provider. For example, in telecommunications, network resources are highly distributed, accessible only through narrow network interfaces. Indeed, the services provided often depend on partner providers, of whom very limited knowledge may be possible.

Another problem in defining negotiation policy is that business strategy can be open-ended and complex. For example, consider pricing strategy. In some industries pricing strategies are very complex. Congestion pricing and revenue management [3] are substantial topics on their own. Designing an easy-to-use, general-purpose negotiation policy language that could express airline ticket pricing policy would be most challenging.

In this paper we present protocol and policy support for negotiation based on a hierarchical negotiation model. A key feature of this model is that a service provider computes offers for a client based on offers returned from lower-level negotiations. There are two cases. In the first, the sub-negotiation is with an external service provider. For example, consider negotiation in home construction. A potential buyer will query a home builder, who in turn queries ‘sub-contractors’ that specialize in electrical work, plumbing, etc.

In the second case of hierarchical negotiation, the lower-level negotiation is with an agent internal to the service provider. Such an agent serves as a kind of gateway or proxy to service provider resources. In other words, local resources and external services are treated uniformly in negotiation as services to be negotiated on. This approach helps in managing the complexity of negotiation strategy, and provides a natural notion of scope to negotiation policy. Each policy component has well-defined interfaces: one for the service it provides, and others for the services it uses.

Note that hierarchical negotiation is distinct from service composition. If a service is composed from other “sub-services”, then one expects negotiation to involve sub-negotiations. However, this negotiation happens at negotiation time, not at the time of service use. Furthermore, to perform hierarchical negotiation one does not need to know how services are composed – only how the terms of sub-services impact the terms of the service. Finally, hierarchical negotiation can be useful even in the absence of service composition, simply as a strategy-structuring mechanism.

Another feature of our model is the use of logic. Models of negotiation typically have buyer and seller exchanging either single bids, or sets of bids. In our model, the queries that clients make to service providers, and the offers that service providers make in return, are captured as logical formulas.

This use of logic gives several benefits. Sets of offers can be concisely expressed, the client need only mention service parameters of interest, and the client can express not only parameter values but also constraints on those values. There are benefits for the service provider, too, because queries can expose client preferences. For example, if a buyer asks
a car seller for a car with at least 200 horsepower, the seller might infer that horsepower is especially interesting to the buyer, and that the buyer prefers cars with high horsepower. Finally, the use of logic allows for a concise policy language, and the leveraging of powerful computational tools.

Based on this hierarchical model of negotiation, we have developed a negotiation policy language and negotiation protocol. The policy language is simple and logic-based. The key idea of the language is that if one knows how to form offers from sub-offers, then one can deduce how to form sub-queries from queries. The negotiation protocol is an application-level protocol that, like AtomPub [4], uses XML over HTTP. Our protocol is simpler than some other existing protocols (such as WS-Agreement [1]), and uses an interface style (a so-called “RESTful API”) that is familiar to many application developers.

The paper is organized as follows. In the following section we describe our model of service negotiation. In Section III we state assumptions we make about services architecture. In Section IV we briefly describe our policy language and its interpretation. In Section V we describe our negotiation protocol. In Section VI we explain how our work relates to the large literature on automated negotiation, and service negotiation in particular.

II. SERVICE NegoTiation MODEL

Service negotiation happens within a larger context that can include service discovery, negotiation, purchase, activation, service use, monitoring, and termination. Our focus is on negotiation, so we assume the existence of support for other phases of service handling. Negotiation is associated with service ordering, provisioning, and activation, and is distinct from service use. One can think of service negotiation as an enhancement of simple service ordering or signup. The distinction between service provisioning and service use is reflected in WS-Agreement [1] as the Agreement and Service layers of its architecture.

Our negotiation model is a variant of two-party, multi-issue negotiation (e.g., see [5]) in which the parties are a client and a service provider1, and the issues are the service parameters. These parameters can include both quality-of-service (QoS) parameters, such as reliability, latency, and price, as well as functional parameters concerning the services behavior. In a storage service the parameters might be the quantity of storage, the reliability of storage, the cost of storage, and the latency in reading and writing to storage.

At a high level, our negotiation model supports the following events: a negotiation request by a client, a query from client to service provider, an offer from service provider to client, and an acceptance of an offer by a client. When a service provider receives the acceptance, it acknowledges the acceptance by returning an invoice. Finally, either party can drop out of the negotiation.

A. Configuration Types

In response to a client’s request to begin negotiation, a service provider will return the service parameters that are negotiable. We capture these parameters as a service configuration type. Roughly, this is a record type giving the names and types of the negotiable terms of the service, plus constraints on values of that type. For example, the configuration type for a storage service might be as follows:

```
schema {
  name: storage,
  type: {
    capacity: int, // in GBytes
    price: decimal // USD per month
    reliability: {low, medium, high}
  }
  constraints: { capacity >= 0, price >= 0 }
}
```

A configuration type is much like a simple schema for a single database relation, and similar to the “negotiation template” of [6]. A value of a configuration type is a tuple that satisfies the constraints. For example:

```
(capacity = 100, price = 5, reliability = high)
```

Note that the parameters of the configuration type are not parameters to be used in run-time invocation of the service through its API. Instead, they are typically provisioning parameters. For example, in the case of a storage service the API would include read and write operations, and the parameters of these operations would include the keys or addresses of the data to be read or written.

B. Expressing Negotiation Moves with Logic.

In some negotiation models, buyer and seller exchange “concrete” offers, which for us would mean a value of a configuration type. For example, an offer for a storage service might be:

```
(capacity = 100, price = 3, reliability = medium)
```

In our model, buyers make queries, and sellers make offers, both of which are expressed as formulas of quantifier and variable-free logic over the configuration type. For example, a query for storage might be:

```
price \leq 3.5 \land capacity = 100
```

This model is a generalization of the model in which they exchange concrete offers. For example, one can represent a small set of concrete offers as a formula:

```
(price = 3 \land capacity = 100 \land reliability = medium) \lor
(price = 3 \land capacity = 200 \land reliability = low)
```

On the other hand, a formula might express a large set of offers:

```
capacity > 1000 \land reliability = low
```

1We often use “buyer” and “seller” rather than “client” and “service provider” when we discuss negotiation in general terms.
In classical negotiation theory (e.g., [5]), a query is understood as expressing a buyer's preferences. In some work on service negotiation (e.g., [7]), the query is understood as expressing requirements. Our model does not fix a position on this issue. For example, there is no requirement in our model that a seller's offers satisfy a buyer's query. For example, in response to query \( \text{price} \leq 3 \land \text{capacity} = 100 \), a seller might respond with \((\text{price} = 2.5 \land \text{capacity} = 100) \lor (\text{price} = 1.5 \land \text{capacity} = 500)\). Indeed, the offers are not even required to be consistent with the query. In the example just given, the seller might respond \( \text{price} = 3 \land \text{capacity} = 80 \). This flexibility helps both buyer and seller when a seller cannot offer exactly what the buyer has asked for.

C. Hierarchy in Negotiation

By “hierarchical negotiation” we mean that a service provider may engage in negotiation with “sub-providers” to determine the offers to be sent in response to a client query. For example, a home builder may negotiate with subcontractors before providing a quote to a potential home buyer.

Fig. 1 shows how a generic service provider might negotiate without hierarchical negotiation. The provider owns resources and offers services that the resources make possible. In making negotiation decisions, the provider will look at its local resources and business operations data, which tracks things such as customer information and buying histories, as well as resources it can purchase from suppliers, and services it can obtain from other suppliers.

Fig. 2 shows how local resources and external providers can be treated uniformly as negotiable services. The “wrappers” used to allow resources to be treated as services can be designed to reveal as much or as little resource state as desired. If external service providers do not support negotiation themselves, negotiation proxies can be developed.

Negotiation on a service need not be supported directly by the service provider. Fig. 3 is a variant of Fig. 2 in which negotiation is supported through a third-party provider. In this scenario negotiation is itself a service, and the negotiation service provider may provide negotiation support for many providers, for example in a services marketplace.

Another element of our negotiation model is how a negotiation and its sub-negotiations are coordinated. When a client initiates negotiation with a service provider, the provider will start any needed sub-negotiations. When a client sends a query to the provider, the provider will form sub-queries for the sub-providers, and send these sub-queries to them. It will then collect offers from the sub-providers, and from them form an offer and send it to the client. The offer will contain, hidden to the client, information about how the offer depends on the collected sub-offers. When a client accepts an offer, the provider will extract the sub-offers from the offer, accept the sub-offers, and return an invoice to the client.

Thus, in our model hierarchical negotiation can alternatively be understood as “nested” negotiation. Viewed as a tree, queries from the root negotiation travel down the tree to sub-negotiations, while offers flow up the tree. Acceptance of an offer at a node in the tree leads to acceptance of offers at child nodes, but possibly also to termination of negotiation at other child nodes. This is in contrast to an approach in which offers to a client are made only after terminating all sub-negotiations.

III. ARCHITECTURE FOR SERVICE NEGOTIATION

In the following two sections we describe a policy language and protocol for service negotiation. We now describe the architectural elements we assume. A client negotiates
with a service provider through a negotiation server, which may or may not be hosted by the service provider. The client interacts with the server using a negotiation protocol. Within the negotiation server, there is a software component that encapsulates the core negotiation decision-making. We call this component the negotiation decision point. It takes a client id, a client query, and previous client queries of the negotiation as input. It produces offers as output.

Fig. 4 shows a refined version of the architecture in which the negotiation decision point is implemented as a negotiation policy engine, which computes offers from client queries according to a declarative negotiation policy. The goal of providing for declarative negotiation policy is to make the service provider’s negotiation strategy more transparent and simpler to modify.

The key elements of our support for negotiation are therefore a negotiation protocol, software support for this protocol (such as language-specific client-side libraries), a negotiation policy language, and a policy engine that interprets policies written in the policy language.

If a service provider cannot express its desired negotiation policy in our policy language, or prefers not to, the provider can write its negotiation decision point in whatever form it prefers. Indeed, a service provider may choose to adopt our negotiation protocol but not have an identifiable negotiation decision point in the software architecture.

IV. A NEGOTIATION POLICY LANGUAGE

In this section we describe a declarative policy language for service negotiation. The language we present is not a general-purpose negotiation policy language; it is designed for machine/machine, hierarchical negotiation. In other words, it is designed for cases in which negotiation on a service depends on negotiation of sub-services. Not all policy can be defined this way; inductively there must be some “base” policies. Furthermore, the interpretation of the language described here is based on the assumption that queries express client requirements, not simply preferences. Other interpretations of a policy in our language are possible.

The key goal of our language design is to make policies as simple and concise as possible. We do not want policy writers to have to specify the queries that should be sent to "sub"-negotiation servers. The job of defining these queries could be tedious and error-prone. So instead we design the policy language so it specifies the high-level information necessary for a machine to compute these queries.

An assumption in our language is that negotiation servers do not discover sub-services dynamically, in the middle of negotiation. Instead, the sub-negotiation servers are bound statically to policy variables at compile time. An analogy can be made with a home builder. In the dynamic case the builder would find subcontractors on-the-fly and them obtain offers from them. in the static case the builder would get offers from a fixed set of "sub"-builders.

Due to space constraints, we do not provide a formal language definition, but instead illustrate the language through simple example policies. More detail on the language design and our policy engine shall be presented in a further paper.

Consider first a service that acts as a storage broker, obtaining storage from other services. A broker of this kind is especially helpful when multiple storage providers exist, with price and availability changing dynamically. For example, some international phone service providers broker surplus capacity from major phone companies.

For simplicity we assume the storage service has only two parameters: capacity (in GBytes) and price (in USD/month for given capacity). We suppose also that the service provider uses only two sub-service providers. The question is: if a query for storage is received by the storage broker, how should the broker query the two storage services, and how should it compute an offer?

Suppose the received query is $\text{capacity} = 400 \land \text{price} < 20$. One strategy for the broker would be to ask the two storage services independently for 400 or more GBytes of storage, and then to simply offer the cheaper one. Another strategy would be to query the storage services sequentially, with the query sent to the second server depending on the offer from the first.

The idea behind our language is as follows. Rather than require a policy writer to define the queries to be sent to sub-providers, we ask the policy writer only to define how offers from sub-providers map to client offers. From this information a policy interpreter can compute the queries to be sent to sub-providers, and how the responses should be combined. We regard this policy style as natural, because we expect a service provider to have intuition about how its service is “built” from sub-services.

Fig. 5 shows one policy a storage service provider might use. In this policy, two sub-storage-providers are asked independently for storage, and their offers combined.

The policy is presented in a JSON-like syntax. It consists of parameters, a name, a type, client preferences, and rules. We now paraphrase the policy in English. The parameters show that two sub-services are used, both with configuration type 'storage'. The policy is for a service of the same type. The 'preferences' value expresses that the client prefers low
values of the price parameter. The policy contains two rules. The first says that, in deriving a client offer, the capacity will be equal to the capacity given in the offer from negotiation server s1, and the price will be s1’s price, but with a 10% markup. The second is the same, except for server s2.

Fig. 6 shows a round of negotiation based on the policy. The client asks for 100 GBytes of storage, at a price of no more than 5 dollars/month. Because of the service provider’s markup, this means that the sub-providers must be asked for 100 GBytes at a price of no more than 50/11 dollars/month. The two sub-providers, s1 and s2, respond using different styles. Provider s1 offers the requested price, while s2 offers what may be its “best” price. The offer returned to the client contains only the lower-priced offer, using the preferences information of the policy. It is not shown in the figure, but one can derive from the policy that the queries to s1 and s2 can be made in parallel.

Fig. 7 shows an alternative policy for storage brokering, again using sub-services s1 and s2. The idea of the policy is that storage will be obtained from both s1 and s2, not just one of them. The policy’s single rule says that the client should be offered the sum of what is offered by s1 and s2, and the price should be 10% higher than the sum of the prices from s1 and s2.
V. A Protocol for Hierarchical Negotiation

In this section we discuss requirements for a negotiation protocol supporting our model, and briefly describe the protocol we developed.

A. Protocol requirements

A key piece of support for our negotiation model is a protocol through which a client and service provider can send their negotiation moves across a network. Our negotiation protocol was designed with several requirements in mind. The most basic of these is that the protocol support our negotiation model. This means the client and service provider should be able to iteratively send queries and offers. Another requirement is that the protocol be generic: not tied to any particular service. Also, service providers should be able to gracefully cope with changes to the configuration types of services they support.

Another class of requirements concern security issues. It must be possible to make offers to one client that cannot be used by any other clients, as service providers may wish to provide “exclusive” offers to special clients. Queries and offers should be confidential, as their content can reveal information about clients and service providers intentions. Finally, clients should not be able to produce counterfeit offers and present them to the service as valid.

We also considered usability requirements. Efficient implementations of the protocol should be possible, the protocol should be easy for clients and service providers to use, and should be inexpensive to adopt for use with existing web services.

These requirements led us to design an application-level protocol for service negotiation that uses HTTP for transport, XML for content, and uses Fielding’s REST architectural style [8]. This is also the approach used in the AtomPub publishing protocol, and many “RESTful” web service APIs. The familiarity that web service developers have with RESTful web service APIs helps immediately with ease-of-use. The similarity of our protocol to web service APIs suggests the idea of “negotiation as a service”, described in [9]. Our protocol is generic because it is parameterized by a service configuration type. The returned element also provides the configuration type.

We describe our protocol following practice in RESTful service design [10]. We identify a collection of resources, identifying each resource with a URI, showing how these resources are interrelated (hyperlinks), and explaining how HTTP methods operate on the resources.

B. Resources

Fig. 9 lists the resources of our API and their URI templates, assuming example.com is the negotiation server URI. The “base” resource is negotiations, the collection of all negotiations. We use secure HTTP to assure the client the identity of the negotiator. Once a secure connection is established, both ends know that queries and offers are transmitted securely.

Another resource is the negotiation. An example negotiation URI is example.com/negotiations/1. At the heart of a negotiation is a history of queries from the client and offers from the server. The list of queries and offers can be retrieved from the queries and offers container resources. Each new query from a client is added to the list of queries, and the offers returned by the service provider is added to the list of offers. A client can retrieve any query or offer from the negotiation server, provided their expiration date has not passed.

A negotiation resource also tracks whether agreement has been reached during the negotiation. An invoice resource is used to record the agreement. This resource contains the negotiated service URI, and the URI of the payment server, if necessary.

C. Hyperlinks

The resources described in the previous section are interrelated. The negotiation resource keeps track of queries and offers. Thus, the resource stores URIs for each query placed and every offer sent back during the negotiation. Each query includes a link to the corresponding offer. Likewise, every offer includes a link to the appropriate query. Finally, an invoice resource keeps a link to the accepted offer. Using these links, programmers can navigate through the query history, check their corresponding offers, and retrieve the invoice and the related offer.

D. Methods

We now explain how HTTP methods are used to perform moves of a negotiation.

Establishing a negotiation: The client sends a secure HTTP POST on the negotiations container URI to initiate a negotiation. The content in the message body is an XML negotiation element specifying a user URI and a service name. The service name is the key to retrieve the service configuration type.

A 201 Created HTTP response indicates successful creation of a negotiation. The body of the response contains an XML negotiation element that includes the configuration type of the service. The returned element also provides URIs for the queries, offers, and invoice. Error responses are also possible. For example, the specified service may not exist.

Making a negotiation query: The client sends a secure HTTP POST on a negotiation’s history URI to place a query. The body of the POST is an exchange element containing a query expressed as a logical formula over the service’s configuration type.

A 201 Created HTTP response indicates a successful query. The body of the response is an offer element containing the query as well as the server’s offers. This element
contains a logical formula over the service’s configuration type. Error responses are also possible, including a 410 Gone response, indicating that the server is terminating the negotiation.

**Accepting an offer:** The client sends a secure HTTP PUT on the negotiation’s invoice to accept an offer. The body of the PUT message is a invoice element, which contains a offer element. The formula of this offer element must determine a value for each parameter of the service’s configuration type.

A 201 Created HTTP response indicates that the server confirms acceptance of the offer. The body of the response contains an invoice element, a URI specifying the server that provides the service, and, optionally, a URI for a payment server. The service URI might include parameters and values. For example, these values could service keys that support security.

**Client termination of a negotiation:** The client performs an HTTP DELETE on a negotiation to breakup a negotiation. The client can end a negotiation provided no offer has been accepted. Similarly, the negotiation server can end a negotiation, as long as no offer has been accepted by the client.

### VI. RELATED WORK

Our negotiation model is similar to the model underlying the SNAP negotiation protocol [7]. In SNAP, negotiation moves are expressed as conditions similar to our logical formulas, the idea of virtualizing resources as services is present, and the value of hierarchy in negotiation is emphasized. However, there our significant differences between SNAP and our work. For example, SNAP uses three forms of SLA, while we use only one. SNAP treats client queries as requirements, while we do not fix this interpretation in our model. SNAP does not describe how a negotiation coordinates with sub-negotiations, while we do. Outside of the negotiation model, we provide an application-level negotiation protocol, and policy support for hierarchical negotiation, while SNAP does not.

The idea of treating resources as services was also proposed in [11]. However, our approach is different in that the same style of interaction – negotiation – is used for working with other service providers and with local resources. In contrast, in [11], the resource “wrapper” is treated as a “resource broker”.

Sierra et al [12] present a model of service negotiation closely based on classical models of two-party, multi-issue negotiation. They define classes of negotiation tactics and show that negotiation converges under certain conditions. Like most work based on theoretical models of negotiation, hierarchical negotiation is not treated. Furthermore, a policy language is not defined.

Amongst negotiation protocols, one of the best known is WS-Agreement [1]. Negotiation in WS-Agreement, unlike our work, is symmetrical: either buyer or seller can initiate negotiation. Furthermore, WS-Agreement has a more developed notion of agreement template. For example, a distinction is made between negotiation terms and service terms. However, a WS-Agreement query is more restrictive than in our model, and, more importantly, the service provider can only accept or reject a client query. Furthermore, in WS-Agreement the configuration type is not sent from service provider to client prior to negotiation properly.

A variant of WS-Agreement closer to our work is described in [13]. There, the configuration type is exchanged at the beginning of negotiation, and the service provider can respond to an “agreement request” with an “agreement offer”.

RNAP [14] is a protocol for negotiation and pricing of QoS in multimedia services. RNAP supports a sort of “transaction” feature, in that a client query specifies requirements for multiple services at once. The negotiation server replies with a quotation message giving price and service parameters for the services.

Compared to work on negotiation protocols, there is little work on negotiation policy languages. Much of the existing work provides support for generic, two-party negotiation. In contrast, our focus is on service negotiation, and on a hierarchical model, which can involve more than two parties. Furthermore, existing work often uses a rule-based language framework.

For example, a rule-based approach is used in [6]. Despite the declarative nature of the rules, the approach is more operational than our own, in that rules describe when offers should be made, what the content of the offers should be, and when an offer should be accepted. Negotiation templates and proposals (our configuration type and offers) are expressed in OWL-Lite. In [15], service negotiation strategy is formalized as a collection of rules in the PANDA policy language[16], which is designed to express preferences. It
is Java-based framework, where users can add new objects and extend existing objects. In the Aspire system [17], Atin negotiation support agents are used to suggest negotiation strategies and offers. Atin’s suggestions are based on preference information and negotiation history.

We suggested in Section I that the logical structure of queries and offers could be used to discover client and service provider preferences. We are interested in how work on the use of machine learning to discover opponent preferences in negotiation (e.g., [18]) could be extended to use this logical structure.

VII. SUMMARY

We have described a hierarchical model of service negotiation based on the use of logic for buyer queries and seller offers, and have defined a negotiation protocol, policy language, and policy engine based on the model. The use of hierarchy in negotiation in natural and has been discussed previously. The developments presented here go beyond existing work in several ways. First, our model of hierarchical negotiation is simple, logic-based, and defines how higher and lower-level negotiations coordinate. Second, our protocol is a modern, application-based protocol with a RESTful design. Finally, our negotiation policy language explicitly supports hierarchical negotiation.

A question for further work is how mathematical models of negotiation, such as bargaining theory, have been or can be extended to support hierarchical, or “nested” negotiations.

ACKNOWLEDGMENT

We thank Lalita Jagadeesan, Alan Jeffrey, and Michael Benedikt for helpful discussion on the material in this paper, and for comments on earlier versions.

REFERENCES


