SCALABLE COMPUTING: PRACTICE AND EXPERIENCE
Volume 7, Number 1, pp. 23–36. http://www.scpe.org

ISSN 1895-1767
© 2006 SWPS

A WS-AGREEMENT BASED RESOURCE NEGOTIATION FRAMEWORK FOR MOBILE AGENTS

D. G. A. MOBACH, B. J. OVEREINDER, AND F. M. T. BRAZIER*

Abstract. Mobile agents require access to computing resources on heterogeneous systems across the Internet. They need to be able to negotiate their requirements with the systems on which they wish to be hosted. This paper presents a negotiation infrastructure with which agents acquire time-limited resource contracts through negotiation with one or more mediators instead of individual hosting systems. Mediators represent groups of autonomous hosts. The negotiation protocol and language are based on the WS-Agreement Specification, and have been implemented and tested within the AgentScape framework.

Key words. mobile agents, resource management, agent-based negotiation, WS-Agreements

1. Introduction. One of the assumptions behind the mobile agent paradigm in open, heterogeneous environments is that agents will have access to computing resources. Little thought has been given to the way in which this can be implemented. Not only do they need access, they need to be able to plan coordinated resource usage across multiple domains. Recently, negotiation of the conditions and quality of service of resource access has been considered to be an important capability for distributed, service-oriented architectures. This paper focuses on the negotiation of resource access for mobile agent applications deployed on Internet-scale, open distributed systems. The resources required by agents can vary from CPU type, bandwidth, to the provision of specific services (e. g., databases, web servers, etc.), and level of security required, depending on the task at hand. Well-defined, open protocols and mechanisms are necessary for agents to negotiate their resource access requirements with heterogeneous hosts.

This paper presents a negotiation infrastructure within which individual agents acquire time-limited contracts for the resources they need, through negotiation with one or more system domain coordinators: mediators representing multiple autonomous hosts. The protocols with which agent applications, domain coordinators, and hosts interact, are based on the WS-Agreement Specification [1] with application dependent domain ontologies for specific resources.

The next sections present the negotiation infrastructure, including the model and the architecture. Section 4 describes a specific implementation of this architecture which is integrated within the AgentScape framework. The application dependent domain ontology for specific computer resources is presented together with examples of the WS-Agreement based protocol. In Section 5, two different policies for request distribution by the domain coordinators are compared empirically and evaluated. The paper concludes with related work and discussion.

2. Negotiation infrastructure. The overall goal and use of the negotiation infrastructure is to allow for the negotiation of terms of conditions and quality of service of resource access by agents. The negotiation model includes the exchange of agreement offers and acceptance of the offers between different parties.

2.1. Design Goals. The negotiation infrastructure has to deal with (i) large numbers of heterogeneous agents, and (ii) dynamic groups of heterogeneous hosts each with their own specific sets of requirements.

From the agent's perspective, the negotiation infrastructure defines a uniform and straightforward negotiation protocol and well-defined interface. Agents are not interested in knowing how the process of allocating specific resources to specific hosts is achieved: their interest is to acquire the resources they need. The negotiation infrastructure needs to hide the details from the agent applications.

On the other side, hosts need to keep full control over their own system, over the use of their resources by agent applications. Negotiation policies spanning multiple hosts, allowing specification of resource access and usage policies over a set of hosts (e.g., for load balancing purposes, or virtual organization-wide policies, etc.) must also be facilitated.

2.2. Negotiation Model. In our negotiation model, *hosts* (H) are autonomous entities that provide *resources* (R) to *agents* (A) under specific usage and access policies. Hosts are aggregated into *virtual domains*. The *domain coordinator* (DC), represents the hosts (H) within a virtual domain in the negotiation process, negotiating with both agents and hosts. Figure 2.1 shows an overview of the model.

*IIDS Group, Department of Computer Science, Vrije Universiteit Amsterdam, de Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands

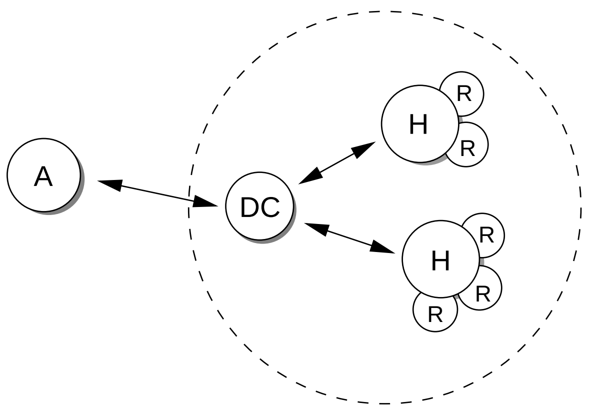


Fig. 2.1. *Negotiation model overview.*

The use of a mediating domain coordinator makes a two-layered negotiation process within the model possible. Agents negotiate resource access with domain coordinators, and domain coordinators, in turn, negotiate with groups of host managers in virtual domains to obtain the actual resources agents require. The results of negotiation are time-limited contracts specifying which resources may be accessed during the time span of the contract, and under which conditions the resources may be used. Agents can negotiate their options with domain coordinators of multiple domains, and select the DC that provides the best offer.

In the model presented in this paper, a domain coordinator represents a virtual organization of resource providers. Agents are unaware of the individual resources behind a domain coordinator: a domain coordinator is viewed by agents to be a single virtual resource provider. The task of selecting one appropriate offer (based on the available resources at a specific point in time) has been delegated to the domain coordinator. Alternatively, a domain coordinator could return a set of possible offers, letting a requesting agent choose the most appropriate. The model presented in this paper supports both options, but only the first is discussed. Section 6 addresses the second option in more detail.

The negotiation protocol and language used in our negotiation model are based upon the *WS-Agreement Specification* [1]. This specification defines the format used to specify agreement descriptions and agreement interactions.[1] The specification defines an XML-based language for agreements between resource providers (hosts) and consumers (agents), and a protocol for establishing these agreements (these agreements are time-limited contracts in our model). Agreement *terms* are used to describe the (levels of) service involved. Two types of terms are distinguished for agreement specifications: (i) *service description terms*, describing the services to be delivered under the agreement, and (ii) *guarantee terms*, expressing the assurances on service quality (e.g., minimum bounds) for the services described in the service description terms. An agreement specification also contains a *context* section, containing meta information about the agreement (see Figure 2.2). This section of the agreement can be used to specify the parties of the agreement, the duration of the agreement, etc. The specification of domain-specific term languages is explicitly left open.

The WS-Agreement interaction model (see Figure 2.3) defines that consumers (C) can request agreements from resource providers (P) by issuing an agreement *request* based on available agreements *templates*, which, if accepted, result in new *agreements.*

In the proposed negotiation model, hosts provide an agreement interface to the domain coordinator. The domain coordinator aggregates the templates offered by the hosts into composed templates. The domain coordinator makes these combined templates available to agents. Agreement requests made by agents are received by the domain coordinator. The domain coordinator negotiates an agreement with the hosts with requested resources.

The interaction protocol as specified in the WS-Agreement Specification only allows for a single "request, accept" interaction, in which the requesting party receives either an *accept* of *reject* message from the providing party as a response to an agreement request. This is a very limited interaction model. In the model proposed in this paper, an additional *accept/reject* interaction sequence is introduced, allowing the requesting party to

[1]This specification is currently under development by the Global Grid Forum's Grid Resource Allocation and Agreement Protocol Working Group.

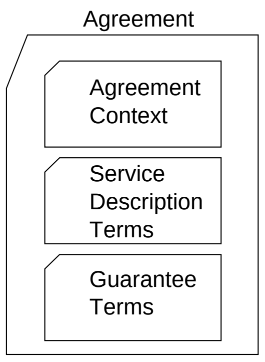


FIG. 2.2. *WS-Agreement contents.*

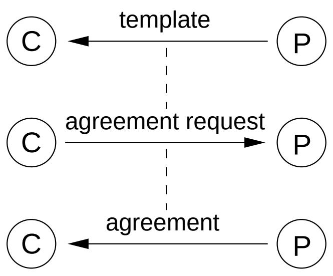


FIG. 2.3. *WS-Agreement protocol.*

explicitly accept or reject an offer created by the providing party. For example, in the context of mobile agent applications, this allows agents to negotiate with multiple domain coordinators simultaneously, and accept the best offer from the set of offers received. Additionally, an explicit *request for templates* interaction is specified. This step in the protocol allows for the initial exchange of information between agents and a domain coordinator, for example for authentication purposes. Figure 2.4 shows the extended interaction model.

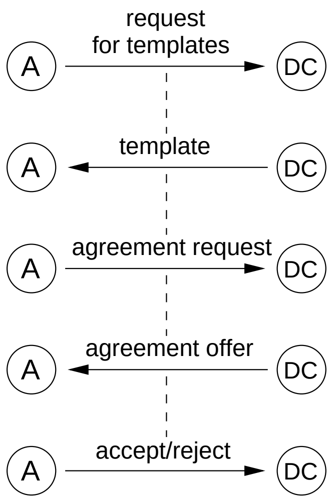


FIG. 2.4. *Extended WS-Agreement protocol.*

3. Negotiation Architecture. The negotiation architecture defines the subsystems and interfaces of the negotiation infrastructure. The two important subsystems host manager and domain coordinator and their interfaces are presented in detail.

3.1. Host Manager. A host manager is responsible for providing and managing resources on its host (see Fig. 2.1). This includes functionality for negotiation, creation, and enforcement of agreements. It is the responsibility of the host manager to translate resource usage and access policies into templates on demand. These templates specify which resources can be made available at a specific point in time. The offer a host makes on request of a domain coordinator is based on these templates. After the negotiation phase, the host manager monitors and controls the resource usage to ensure that agreements are honored.

Figure 3.1 shows the architecture and negotiation interface of a host manager. The agreements in the model are time-limited contracts: agreements that expire after some predetermined time. In the presentation of the architecture, the term *lease* is used instead of time-limited contract. Each host manager is equipped with three modules: a *leasing module*, implementing the main negotiation functionality; a *policy manager* containing *resource policies*, which are applied by the leasing module; a *resource manager* with *resource handlers*, allowing monitoring and control of resource access. The components of the host manager shown in Fig. 3.1 are further described below.

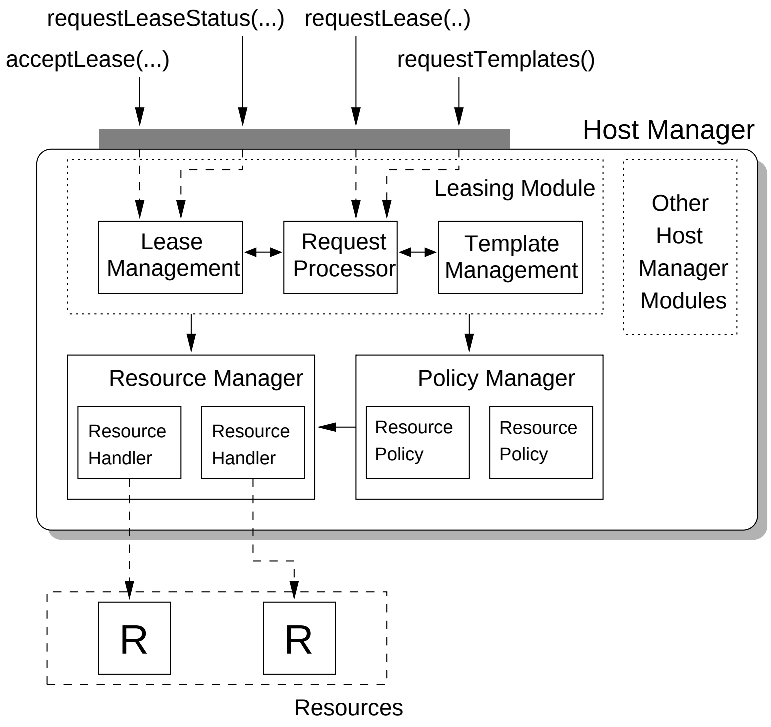


FIG. 3.1. *Components within the Host Manager.*

3.1.1. Leasing Module. The leasing module in the host manager implements the negotiation and agreement protocol. The functionality of the leasing module is available via the interface of the host manager.

Leasing Interface. The leasing interface offered by host managers to their location manager contains the following calls:

- `requestTemplates(): template-list`
  Request the available lease templates.
- `requestLease(LeaseRequest): lease`
  Request a lease based on the supplied lease request.
- `acceptLease(LeaseID)`
  Accept a lease. Returns the accepted lease document.
- `requestLeaseStatus(LeaseID): lease`
  Request the current status of a lease. Returns a lease document, including the current status of each term.

Request Processor.

- Responding to template requests from the domain coordinator according to local policies.

- Creating lease offers. This involves determining the availability of the requested resources, and creating offers based on the incoming request, resource usage and access policies, and the current status of the resources.

Template Management.

- Creating templates based on available resources, resource usage, and access policies, and actively maintaining this information. Note that policies can be dynamic, that is, change over time (e.g., half of available capacity can be reserved during office hours, complete capacity is available outside office hours).

Lease Management.

- Enforcing the accepted leases. This involves ensuring that the resource manager module performs the required resource negotiation tasks.
- Handling expiration of leases. This involves freeing the resources specified in the expired lease, and possibly sending notifications of lease expiration to the domain coordinator.
- Maintaining lease offers: removing the offers after a certain set time, or implementing the offer after notification of acceptance has been received.
- Handling requests for status information on the running leases.
- Handling violation of leases. In cases where resource usage cannot be strictly enforced, and only monitoring can be performed, lease violations should be handled. When an application violates the conditions set in a lease, appropriate actions should be performed, such as suspending or killing the violating agent.

3.1.2. Policy Manager. The policy manager module contains resource policy descriptions which can be used by the leasing module during the processing of requests. Policies can be defined for specific resources, or policies can be defined covering other aspects of incoming requests (identity of the requesting application, or "global" host policies such as the total number of requests, etc.). A resource policy can contain static information, such as the maximum number of allowed requests for a resource, but can also refer to the monitoring capabilities of resource handlers to incorporate up-to-date monitoring data concerning the resources to which the policy applies.

3.1.3. Resource Manager Module. The resource manager module contains a set of resource handlers, enabling the leasing module to manage resources available on the host. Each resource at a host is represented by a resource handler. The handler implements a resource independent interface for the leasing module to monitor and control the resources. Each resource handler supports: (i) creation of resource reservations based on lease offers; (ii) implementation of the reservation, which activates the resource handler to start monitoring resource consumption with respect to accepted leases; (iii) release of a reservation, freeing the resource (amount) related to expired or violated leases. Each resource handler also supports a monitoring interface, allowing for retrieval of resource specific monitoring information, to be used in, for example, resource policies.

- `reserve(LeaseRequest): ReferenceID`
  Can be used to reserve a resource (amount) for a specific lease request. The resource handler inspects the request, and creates a reservation. A reference identifier is returned to enable further management of the reservation.
- `implement(ReferenceID): void`
  Used to request implementation of a reservation (indicated by `ReferenceID`).
- `release(ReferenceID): void`
  Release an implemented resource reservation (indicated by `ReferenceID`).
- `getStatus([ReferenceID]): status`
  Used to request the status of a reservation. Returned value can be one of: *initialized*, *reserved*, *active*, *violated*.
- `getMonitorValue(SensorID): domain_specific_value`
  Used to request resource specific monitoring information concerning a resource.

3.2. Domain coordinator. The domain coordinator abstracts from the individual hosts (resource providers) and presents the aggregated resources as one virtual resource provider. The domain coordinator is responsible for resource access negotiation with applications and its enforcement. To this purpose it provides applications with templates of resources available within its domain at the time requested. The domain coordinator, in turn, requests and receives information on availability of resources from its hosts, and combines this information if, and when appropriate, to construct application directed templates.

Once a template-based request is received from an application, the domain coordinator pursues delegation of resources to hosts. Upon receiving the host bids, the domain coordinator chooses based on available templates, host and domain policies, and returns a proposed lease if possible. If a proposed lease is accepted, the domain coordinator is responsible its effectuation and enforcement.

Figure 3.2 shows an overview of the leasing module within the domain coordinator.

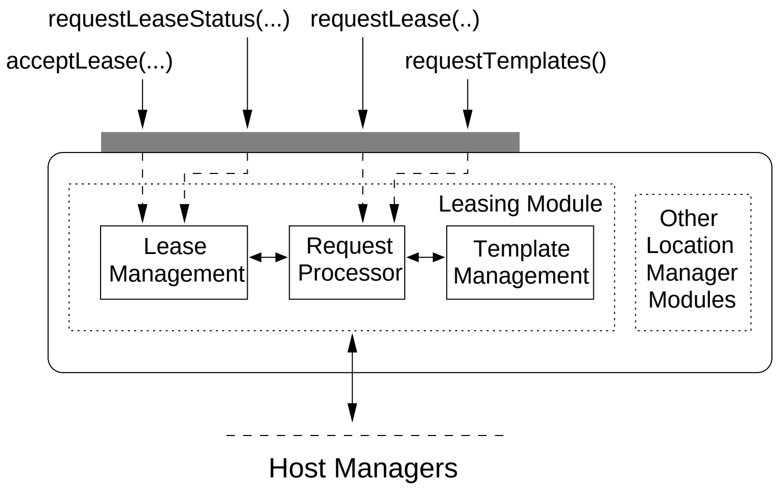


Fig. 3.2. *Leasing components within the domain coordinator.*

Request Processor. This component is responsible for the following tasks:

- Processing requests for templates by applications. This implies checking policies to determine to which template information the application is entitled.
- Processing requests for leases by applications. This involves determining whether the request is based on a valid template, and whether the request exceeds the bounds set by that template.
- Handling lease offers returned by hosts in response to requests. If more than one host was sent the same request, a choice has to be made between their offers. In addition, if the offers are part of a request based upon a combined template, the offers are combined into a single offer for the application. Further, when a lease proposal is accepted by an application, the hosts offering the lease are informed of acceptance.
- Determining from which hosts offers are requested. This involves determining which host(s) are offering relevant templates, and possibly splitting the request into multiple requests for different hosts, if a combined lease template was used by the application.

Template Management. This component requests, creates and maintains information about the templates on which leases are based. This component performs the following tasks:

- Obtaining and maintaining template information of the hosts currently in the domain.
- Creating template combinations of resources from multiple hosts in a single template. This involves applying local template policies specifying which host templates can or cannot be combined.

Lease Management. The lease management component maintains information about leases, lease requests made by applications, and lease proposals from hosts, and performs the following tasks:

- Maintaining status information of current valid leases. This involves actively or passively retrieving lease status information from the hosts responsible for enforcing the leases acting appropriately upon lease expiration.
- Maintaining information of currently outstanding lease proposals.

4. AgentScape Negotiation Architecture. The negotiation architecture described above has been implemented in the AgentScape framework, a framework for heterogeneous, mobile agents. This section describes how the subsystems have been instantiated, and provides examples of how the agreement-based negotiation is used to create leases for agent applications using the AgentScape middleware.

4.1. AgentScape. The AgentScape middleware [8] consists of two layers. At the base of the middleware is the *kernel*, offering low-level secure communication between middleware processes, and facilities for secure agent mobility. On top of the AgentScape kernel, middleware processes provide higher-level middleware functionality to agents. For example, *agent servers* provide a run-time environment for agents, and a *Web service gateway* provides agents the ability to communicate with web services using the SOAP/XML protocol. In AgentScape, virtual domains are called *locations*. An AgentScape location consists of one or more hosts running the AgentScape middleware, typically within a single administrative domain.

In addition to the middleware processes described above, each host has a *host manager* middleware process. This process is responsible for managing the middleware components running on the host, and implementing the required negotiation functionality as described in the architecture. Furthermore, each AgentScape location runs a *location manager* process on one of the hosts, which implements management functionality required for managing AgentScape hosts, and which implements the functionality of the domain coordinator, enabling agent application to enter into resource negotiations with locations. Figure 4.1 shows an overview of an AgentScape location.

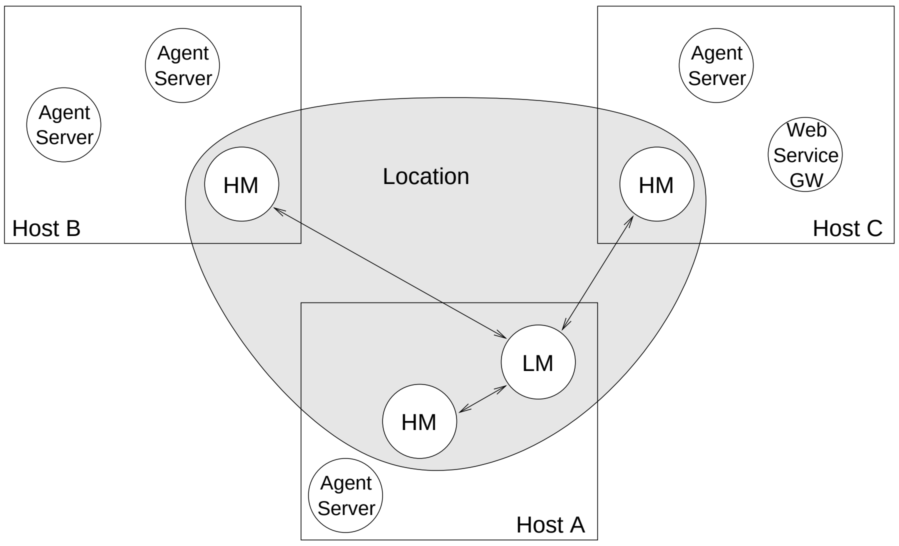


Fig. 4.1. *Overview of an AgentScape location.*

4.2. AgentScape Negotiation Architecture. Within AgentScape, agents can start negotiations with a number of locations, and given the offers the locations provide, select the location offering the best options. The agent then migrates to the location with which agreement has been reached.

4.2.1. AgentScape resources. The AgentScape negotiation architecture defines a set of resources that can be allocated and used by agents in the AgentScape specific ontology. This ontology is used during negotiation. Currently, the following resources are included in this ontology:

- CPU time: The time (in milliseconds) that an agent spends on an agent server.
- Communication bandwidth: The number of bytes/second that an agent may send to other agents.
- Memory: The amount of RAM an agent may consume while running on an agent server.
- Web service access: The web services that an agent is allowed to access using the AgentScape Web Service Gateway.
- Web service call rate: The number of calls that an agent is allowed to do on a web service using the gateway.
- Disk space: The amount of disk space an agent is allowed to use while running on an agent server.

Additional resources can be defined in the future, as the functionality offered by AgentScape is extended. The resources are specified in the XML Schema language, enabling the use of these definitions within the agreement-based negotiation sequence. As an example, consider the three resources specified in Example 4.1. In this example, the `time-on-cpu` resource and the `communication-bandwidth` resource are defined as simple integer values representing the number of milliseconds and the number of Kilobytes/second respectively. The `web-service-access` resource is defined as a list of service names (strings) representing the list of services which may be accessed.

```
<xsd:simpleType name="time-on-cpu"
                type="xsd:positiveInteger" />

<xsd:simpleType name="communication-bandwidth"
                type="xsd:positiveInteger" />

<xsd:complexType name="web-service-access">
  <xsd:all>
    <xsd:element name="service-name" type="xsd:string"
                 minOccurs="1" maxOccurs="unbounded"/>
  </xsd:all>
</xsd:complexType>
```

Example 4.1
AgentScape resource definitions.

The AgentScape specific language is used within the lease model to express resource requirements and usage conditions. In Example 4.2, an example of agent resource requirements is shown. In this example, an agent requests 50 seconds of CPU time, and 50 Kb/s of communication bandwidth.

```
<!-- requirement: 50 seconds CPU time -->
<agentscape:time-on-cpu>
  50000
</agentscape:time-on-cpu>

<!-- requirement: 50Kb/s bandwidth -->
<agentscape:communication-bandwidth>
  51200
</agentscape:communication-bandwith>
```

Example 4.2
Agent resource requirements.

4.3. AgentScape Host Manager. The AgentScape host manager is responsible for offering resources to the location manager. Based on its own information on the status of its resources, and its own policies regarding these resources, the host manager creates a set of templates. Example 4.3 shows an example of a template, using the syntax as defined in the WS-Agreement Specification. The template specifies that this host can now offer two resources, each with specific access conditions. For the first resource: the `time-on-cpu` resource, a maximum value of 100 seconds is specified. The second resource, `communication-bandwidth`, is not restricted by the template.

4.4. Location Manager. The location manager enters into negotiation with host managers within its location on behalf of agents. The location manager maintains information on the templates offers by each of the hosts within the location, and uses this information to provide templates to agents. Agents base their requests for leases to the location manager on these templates. As an example, consider the following request, in which an agent requests a location for 50 seconds of CPU time, and 50 Kb/s of communication bandwidth.

To meet lease requests by agents, the location manager enters into negotiation with the relevant hosts in its location (those that can provide the resources requested). For each request received from an agent, one or more suitable hosts are selected (based on their templates). Each of the hosts then creates an offer based on the current resource conditions. The location manager selects one of the offers, and discards the others, or combines a number of offers into a composed offer. The selected offer is returned to the agent. As mentioned in Section 2.2, multiple offers can be returned to the agent, but does not comply with the AgentScape model.

In the following example, a location manager has received a request from an agent, and has selected two hosts within its location to which it forwards the request. The hosts determine if and to which extent the request can be fulfilled, and return their offers (proposed leases) to the location manager. In Example 4.5, Host 1 returns a proposal in which the requested CPU-time is unchanged with respect to the request from

```
<wsag:Template>
  <wsag:Name>Template1</wsag:Name>
  <wsag:Context/>
  <wsag:Terms/>
  <wsag:CreationConstraints>
    <wsag:Item>
      <wsag:Location>//wsag:ServiceDescriptionTerm//
         agentscape:time-on-cpu
      </wsag:Location>
      <xs:maxInclusive xs:value="100000">
    </wsag:Item>
    </wsag:Item>
      <wsag:Location>//wsag:ServiceDescriptionTerm//
            agentscape:communication-bandwidth
      </wsag:Location>
    </wsag:Item>
  </wsag:CreationConstraints>
</wsag:Template>
```

Example 4.3
AgentScape resource template.

```
<wsag:AgreementOffer>
  <wsag:Name>Offer1</wsag:name>
  <wsag:Context>
    <wsag:AgreementInitiator>
      agentX
    </wsag:AgreementInitiator>
    <wsag:TemplateName>
      Template1
    </wsag:TemplateName>
  </wsag:Context>
  <wsag:Terms>
  <wsag:All>
    <wsag:ServiceDescriptionTerm
          wsag:Name="TimeOnCPU"
          wsag:ServiceName="LocationY">
      <agentscape:time-on-cpu>
        50000
      </agentscape:time-on-cpu>
    </wsag:ServiceDescriptionTerm>
    <wsag:ServiceDescriptionTerm
          wsag:Name="Communication"
          wsag:ServiceName="LocationY">
      <agentscape:communication-bandwidth>
        51200
      </agentscape:communication-bandwidth>
    </wsag:ServiceDescriptionTerm>
  </wsag:All>
  </wsag:Terms>
</wsag:AgreementOffer>
```

Example 4.4
Lease request made by agent.

the agent, and communication-bandwidth is decreased to 10 Kb/s. Host 2 also returns a proposal in which the requested `time-on-cpu` is reduced to 40 seconds, and `communication-bandwidth` is decreased to 30 Kb/s. Also, an `ExpirationTime` element is added to the context section of the proposal, indicating when the lease will expire, if accepted by the agent. Host 1 defines an expiration time of 23:04:44 upon which it no longer guarantees the requested resources, and Host 2 defines an expiration time of 23:10:00.

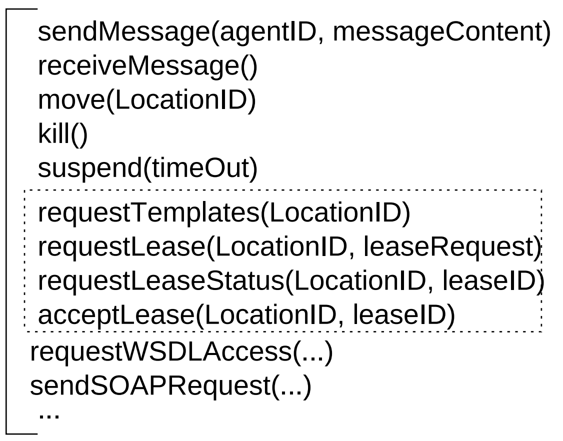
The proposals are received and compared by the location manager. Host 1 offers fully the requested `time-on-cpu`, but offers a `communication-bandwidth` which is substantially lower than the requested bandwidth. The offer made by Host 2 offers a lower `time-on-cpu` value, but does offer a bandwidth value which is

```
<wsag:Agreement>
  <wsag:Context>
    <wsag:AgreementInitiator>
      AgentX
    </wsag:AgreementInitiator>
    <wsag:AgreementProvider>
      Host1
    </wsag:AgreementProvider>
    <wsag:ExpirationTime>
      2005-07-23T23:04:00
    </wsag:ExpirationTime>
  </wsag:Context>
  <wsag:Terms>
    <wsag:All>
      <wsag:ServiceDescriptionTerm
            wsag:Name="TimeOnCPU"
            wsag:ServiceName="LocationY">
        <agentscape:time-on-cpu>
          50000
        </agentscape:time-on-cpu>
      </wsag:ServiceDescriptionTerm>
      <wsag:ServiceDescriptionTerm
            wsag:Name="Communication"
            wsag:ServiceName="LocationY">
        <agentscape:communication-bandwidth>
          10240
        </agentscape:communication-bandwidth>
      </wsag:ServiceDescriptionTerm>
    </wsag:All>
  </wsag:Terms>
</wsag:Agreement>
```

```
<wsag:Agreement>
  <wsag:Context>
    <wsag:AgreementInitiator>
      AgentX
    </wsag:AgreementInitiator>
    <wsag:AgreementProvider>
      Host2
    </wsag:AgreementProvider>
    <wsag:ExpirationTime>
      2005-07-23T23:10:00
    </wsag:ExpirationTime>
  </wsag:Context>
  <wsag:Terms>
    <wsag:All>
      <wsag:ServiceDescriptionTerm
            wsag:Name="TimeOnCPU"
            wsag:ServiceName="LocationY">
        <agentscape:time-on-cpu>
          40000
        </agentscape:time-on-cpu>
      </wsag:ServiceDescriptionTerm>
      <wsag:ServiceDescriptionTerm
            wsag:Name="Communication"
            wsag:ServiceName="LocationY">
        <agentscape:communication-bandwidth>
          30720
        </agentscape:communication-bandwidth>
      </wsag:ServiceDescriptionTerm>
    </wsag:All>
  </wsag:Terms>
</wsag:Agreement>
```

Example 4.5
Host lease proposals.

closer to the requested value than the offer of Host 1. The location manager makes a selection between these offers based on current selection policies, and communicates this offer to the agent. In our example, the location manager chooses the proposal made by Host 2. The agent chooses to accept the offer. After acceptance, the agent has a limited time in which it must migrate to the target location, or the lease offer will expire. After the arrival of the agent at the target location, the agent is allowed to consume the agreed upon resources until the lease expires.



```
sendMessage(agentID, messageContent)
receiveMessage()
move(LocationID)
kill()
suspend(timeOut)
requestTemplates(LocationID)
requestLease(LocationID, leaseRequest)
requestLeaseStatus(LocationID, leaseID)
acceptLease(LocationID, leaseID)
requestWSDLAccess(...)
sendSOAPRequest(...)
...
```

Fig. 4.2. *Lease related calls on the AgentScape agent interface.*

4.5. AgentScape Agent Interface. The interface presented to agents by the AgentScape middleware contains several lease related calls, as shown in Figure 4.2. These calls enable agents to enter into resource lease negotiations with AgentScape locations.

5. Experiments. To evaluate the implementation and assess the operation of the negotiation architecture described above, several experiments have been performed. The first set of experiments centered on the ability of the negotiation architecture to accommodate domain-wide resource policies. The second set of experiments focused on the use of the negotiation architecture to apply “quality of service” policies using individualized host policies.

5.1. Experimental setup. A distributed AgentScape location is set up consisting of nine hosts. Eight hosts are configured to run a host manager and an agent server, and one host is configured to run a location manager. The location manager implements the domain coordinator negotiation functionality. In each of the experiments, agents migrate to the location after a lease has been acquired through negotiation with the location manager. The hosts used for the AgentScape location are part of the DAS-2 cluster at the Vrije Universiteit Amsterdam, consisting of Dual Pentium-III nodes connected by Fast Ethernet (Myrinet-2000 is available between machines at each cluster, but was not used in these experiments). The agents are inserted from a host outside the DAS-2 cluster, also connected by Fast Ethernet.

In the experiments, CPU-time is the main subject of the negotiation process. In each experiment, one thousand agents are inserted into the location. For each agent, a “desired” CPU-time amount is generated according to the Weibull distribution (scale = 3.0, shape = 2.0, mean = 26.587 seconds). This value from the distribution is then used to create a lease request which is then sent to the location. The intervals between lease requests of individual agents are distributed according to the Poisson distribution (mean = 2 seconds). Each lease request received by the location manager is translated into lease requests to the 8 host managers within the location. Each host manager then responds with a lease offer if the requested value is in line with the local CPU-time policy, or responds with an empty offer if the requested value is not in line with the policy. In the experiments, the load on a host is represented as the number of agents running on a host, measured at one second intervals.

5.2. Domain-wide negotiation policy experiments. In the area of distributed systems it is useful to apply domain policies facilitating the distribution of computational load across available hosts in the environment. Two straightforward types of policies are based on the principles of: (1) time-division, in which computational load is scheduled for execution at different times, and (2) space-division, in which computational load is scheduled on different hosts. In these experiments, a round-robin (space-division) negotiation policy is applied, i. e., a location manager collects offers made by the hosts, and applies a round-robin load balancing policy to select one of the offers made by the hosts. This offer is then sent back as an answer to the original lease request. After acceptance of the lease, an agent is inserted at the host that has been selected during negotiation. The agent will then start to consume CPU-time by performing predefined calculations. When the CPU-time delegated to the agent in the lease is consumed, the agent is stopped and removed from the host. In this experiment, hosts are configured with a negotiation policy dictating that all lease requests should be accepted, regardless of the requested CPU-time value. The location manager selects host manager offers according to a round-robin policy, with the aim of to distribute all agents evenly throughout the location.

As a measure for the balance of the load within the AgentScape location, the “Load Balance Metric” is used, as described by Bunt and Eager [4]. This metric is defined by taking the weighted average of peak-to-mean server load ratios. This ensures that a larger imbalance during high-load situations has a greater effect on the LBM measure than a smaller imbalance during lower-load conditions. The value of the LBM measure ranges from the number of servers (8 hosts in the experiments) to 1, where a lower value represents a higher balance (LBM value 1 means perfect load balance). In Fig. 5.1, the LBM values are graphed, calculated over 10 second intervals. The figure shows that a consistent balance is achieved within the location using the round-robin policy, during the insertion of agents as described in the experimental setup. At the end of the experiment, load balance can no longer be enforced, as all agents have been inserted and load imbalance is induced by the completion of agents at a host, while a fraction of the hosts is still executing long running agents. This is shown in the graph by the sharp increase of the LBM value.

5.3. Differentiated host policy experiments. In the second set of experiments, negotiation policies were applied to implement a quality of service policy aimed at improving responsiveness for agents with a relatively short running time (below the mean value as described above). In the experiments, two different host policies are used: a policy allowing only requests below the mean CPU-time value, and a policy allowing only requests above the mean CPU-time value. (The CPU-time values are taken from the same Weibull distribution as described in Section 5.1.) In each experiment, the number of hosts accepting below-mean and above-mean is varied. The round-robin policy of the location manager is still applied, but within the two host groups

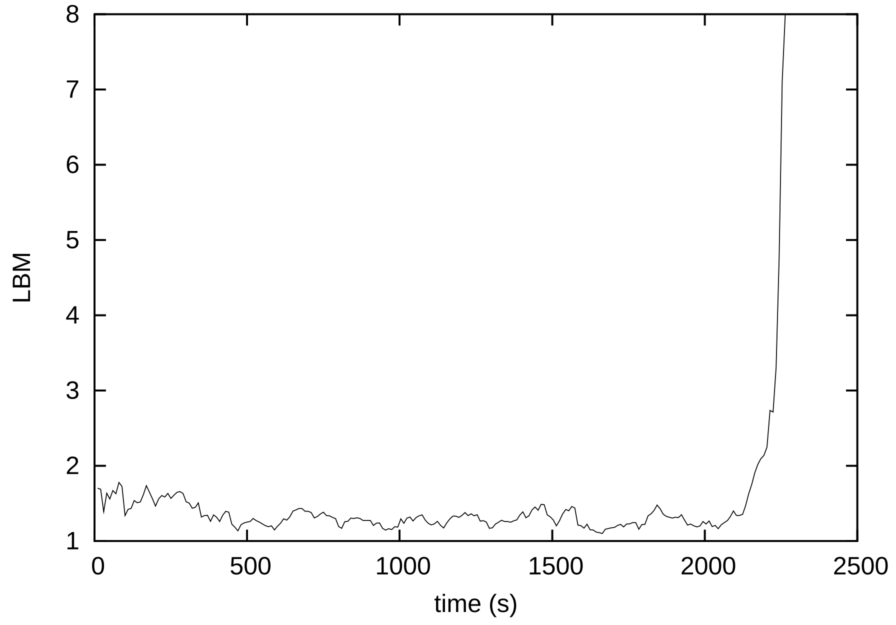


Fig. 5.1. *LBM over 10 second intervals using round-robin negotiation policy.*

separately, as attaining a balanced load within groups is still desirable, but is not feasible across the different groups.

In Table 5.1, the results of these experiments are shown. In the first column, the number of hosts accepting only agents with a CPU-time value below the mean is given. The second and third column present a quality of service percentage for agents with a below-mean and above-mean CPU-time value respectively. The quality of service percentage metric is defined as the actual CPU-time agents have consumed divided by the "wall clock" time agents have spent on a host. The results in the table are the mean over three experiments. A high quality of service percentage of 100 % indicates a perfect quality of service where the resource is completely available to the agent (the agents in the experiments are CPU bound, and, e.g., not waiting for I/O or network communication). A low quality of service percentage means that the agent has to compete with other agents (or generally tasks) to access the resources.

The values in the bottom row are obtained from the load balancing experiments presented in the previous section, in which no differentiation was made based on CPU-time values, and agents could be placed on all hosts. This can be seen as a "reference" value, indicating the responsiveness in the undifferentiated case. From the results it can be argued that a configuration with 8 hosts, where 3 hosts accepting only agents with below-mean CPU-time values (and consequently 5 hosts accepting only above-mean CPU-time), gives agents with a shorter running time a better responsiveness, at a not too great expense for the longer running agents. For 4 hosts reserved for short running agents, the responsiveness dramatically improves with about a factor of 5 compared to the reference results, while the long running agents experience an increased turnaround time of a factor of 1.7.

The experiments have shown that different policies can be relatively easily enforced, both on aggregate location level, enforcing a round-robin load balancing policy, as well as on individual host level, accepting either short or long running agents. It should be stressed that the experiments are not intended to show the performance of specific policies, but rather show how different policies defined on location and host level can be defined and enforced by the resource negotiation infrastructure presented in this paper.

6. Related Work and Discussion. The negotiation architecture described above hides the complexity of managing access and usage of heterogeneous and distributed resources from agents, by providing a uniform negotiation infrastructure aggregating the resources within a virtual domain. The architecture uses the WS-Agreement emerging Grid standard as a basis for its negotiation protocol and language.

The WS-Agreement framework offers an extensible basis for resource management involving distributed heterogeneous resources and distributed applications. In its current state however, the WS-Agreement frame-

# below mean hosts	avg. for below mean agents %	avg. for above mean agents %
2	8.3	38.6
3	24.9	13.2
4	76.3	9.7
5	87.7	5.8
6	90.9	4.5
reference	14.5	16.2

Table 5.1
Quality of service percentage results of the CPU-time differentiated host policy experiments.

work has a number of shortcomings. First, the specification only provides for a basic negotiation protocol and related information structures. This could be sufficient for use in service-oriented environments for which the model is intended, however, in a self-managing application domain, as described in this paper, more elaborate negotiation facilities could provide these applications with more control over allocation and use of resources. Second, the framework does not provide a model describing how enforcement of agreements is to be integrated in the system providing the resources. Although it can be argued that much of this is very domain-specific and cannot be captured in a useful model, the framework could present an abstract model of the required information structures and design of an agreement-based infrastructure supporting the WS-Agreement framework.

In this paper, an extension of the WS-Agreement negotiation protocol is proposed. The addition of an explicit accept/reject interaction sequence allows agents to enter into negotiations with multiple providers and compare received offers. The proposed framework is implemented in the AgentScape middleware. In a recent paper, Paurobally and Jennings [9] also recognize the need for more complex negotiation patterns other than possible within the WS-Agreement Specification. In their paper, richer message types (i. e., *inform* and *bid*) and interaction protocols are proposed in the form of an additional layer, allowing for the specification of agent interaction protocols on top of the WS-Agreement messaging layer. The Grid Resource Allocation Agreement Protocol (GRAAP) working group also extended their work on WS-Agreement with the WS-Agreement Negotiation Specification [2]. Here, a negotiation layer is defined to be incorporated on top of the WS-Agreement Specification. The negotiation layer allows to express negotiation offers in terms expressed in the meta-language already defined in WS-Agreement.

Independent from the WS-Agreement Specification activities, Hung *et al.* [6] proposed a Web service negotiation model called WS-Negotiation. Also, a service level agreement (SLA) template model is presented, with different domain specific vocabularies for supporting different types of negotiation. The negotiation protocol in their model is geared toward integrative negotiation, where both parties locate and adopt the option that provide greater joint utility to the parties taken collectively. The message types reflect this negotiation model and is more extended than the models presented by Paurobally and Jennings [9] and the GRAAP working group [2].

IBM's Cremona [7] (Creation and Monitoring of Agreements) is an effort to create an architecture and set of libraries that implement the WS-Agreement interfaces and agreement (template) management, and provide agreement functionality suitable for implementations in domain-specific environments. The Cremona architecture specifies domain-independent and domain-specific components required for agreement-based management, and the Cremona libraries provide implementations of the agreement interfaces, domain-independent components, and well-defined interfaces for the domain-specific components. Cremona is currently being offered as a part of IBM's Emerging Technologies Toolkit.

The design goals and the realization of the WS-Agreement-based negotiation infrastructure presented in this paper and the Cremona architecture are quite similar. However, the WS-Agreement-based negotiation infrastructure extends the Cremona architecture with the option to combine templates and agreements from multiple resources. The combination of templates and agreements is necessary to accomplish resource aggregation, for example, to implement virtual organizations where multiple resource cooperate to provide a (number of) services.

The concept of leasing has been used in the area of distributed application frameworks, for example in Jini [10], where leases are used for distributed garbage collection. In the Jini framework, clients lease resource access, such as for example service registration within a lookup service. The acquired lease allows a client to make of use of that resource for a limited time-period. When a lease expires, and no explicit renewal is requested by the client (for example because of network failure), the associated resource is made available for other clients, preventing unnecessary resource allocation. This characteristic has been included in the negotiation model presented in this paper. The Jini specification, however, does not cover a negotiation model or protocol specification. In the SHARP [5] architecture, tickets (soft resource claims) can be redeemed by resource consumers for leases (hard resource claims), which guarantee access to a resource. Ticket holders can delegate resources to other principals by issuing new tickets. The goals of the SHARP architecture and the AgentScape negotiation architecture are similar in nature, with the AgentScape negotiation architecture being more oriented towards agent applications.

The focus of our current and future work includes extending the architecture and model with agent level components, allowing application developers to more easily integrate and implement resource negotiation interactions into their applications. As an example, for the AgentScape middleware, a WS-Agreement based Agent Communication Language would enable agents to more easily communicate with the resource negotiation infrastructure. Furthermore, the addition of more expressive and flexible negotiation protocols would allow both applications and resources more fine-grained control of the negotiation process.

As stated in Section 2.2, the current implementation of the domain coordinator in the negotiation infrastructure returns one offer in reply to an agent request. This is an implementation decision and not a limitation of the negotiation model or protocol. If the domain coordinator returns multiple offers, the requesting agent can decide which offer is most appropriate to complete its current task, e. g., considering expected computing time, execution costs, security level, or other uses of resources. Part of the extended negotiation protocol can be the specification by the agent whether it opts for a light-weight negotiation protocol with single offers, or a more complex negotiation protocol with multiple offers.

The negotiation architecture makes it also possible for a virtual provider to check an agent's credentials before even starting to negotiate with an agent. As identity management is an important aspect in the design of large-scale open agent systems [3], this aspect is currently being further explored, in particular in relation to legal implications of the use of mobile agents.

REFERENCES

[1] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, T. Nakata, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, *Web services agreement specification (WS-Agreement) (draft)* 2006, `https://forge.gridforum.org/projects/graap-wg`

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu, *Web services agreement negotiation specification (WS-AgreementNegotiation) (draft)* 2004, `https://forge.gridforum.org/projects/graap-wg`

[3] F. Brazier, A. Oskamp, J. Prins, M. Schellekens, and N. Wijngaards, *Anonymity and software agents: An interdiscplinary challenge*, AI & Law, 1-2 (2004), pp. 137–157.

[4] R. B. Bunt, D. L. Eager, G. M. Oster, and C. L. Williamson, *Achieving load balance and effective caching in clustered Web servers*, in Proceedings of the 4th International Web Caching Workshop, San Diego, CA, Apr. 1999, pp. 159–169.

[5] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat, *SHARP: An architecture for secure resource peering*, in Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY, Oct. 2003, pp. 133–148.

[6] P. C. K. Hung, H. Li, and J.-J. Jeng, *WS-Negotiation: An overview of research issues*, in Proceedings of the 37th Hawaii International Conference on System Sciences (HICSS'04), Big Island, Hawaii, Jan. 2004, pp. 33–42.

[7] H. Ludwig, A. Dan, and R. Keaney, *Cremona: An architecture and library for creation and monitoring of WS-Agreements*, tech. report, IBM Research Division, June 2004.

[8] B. J. Overeinder and F. M. T. Brazier, *Scalable middleware environment for agent-based Internet applications*, in Proceedings of the Workshop on State-of-the-Art in Scientific Computing (PARA'04), Copenhagen, Denmark, June 2004, pp. 675–679. Published in Applied Parallel Computing, LNCS 3732, Springer, Berlin, 2006.

[9] S. Paurobally and N. R. Jennings, *Developing agent Web service agreements*, in Proceedings of the IEEE/WIC/ACM International Conference on Intelligent Agent Technology, Compiegne, France, Sept. 2005, pp. 464–470.

[10] J. Waldo, *The Jini architecture for network-centric computing*, Communications of the ACM, 42 (1999), pp. 76–82.

Edited by: Shahram Rahimi, Raheel Ahmad
Received: November 8, 2005
Accepted: March 19, 2006