Multi-Agent Systems and Grid Services: Towards Robust Continuous Distributed Problem Solving

Yolanda Gil and Varun Ratnakar

USC Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292
gil@isi.edu, varunr@isi.edu

Abstract

Web services are an increasingly viable paradigm for agent-based systems. This paper discusses the advantages of using grid services, an extension to web services that supports robust distributed computation. Our discussion is based on an implementation of information agents using the Open Grid Services Architecture (OGSA), an emerging standard for grid computing that is available in the newer version of the widely used Globus toolkit.

Introduction

Current research on semantic web services focuses on developing expressing languages for describing services as extensions to the WSDL (Web Service Description Language [Christensen et al 01]) standard. DAML-S [Ankolenkar et al 02] and OWL-S incorporate complex process and plan description constructs as well as terminological descriptions of domain terms in DAML and OWL respectively. Other languages describe complex compositions of web services, such as WSFL [Christensen et al 01], XLANG [Thatte 01], BPML, BPSS, BPEL4WS, and WSCL [Banerji et al 02]. All these languages build on WSDL to address semantic-level issues of service competence, behavior, and coordination.

A very important requirement for robust intelligent distributed problem solving is robustness in face of dynamic execution environments. We argue that grid services provide a much better substrate for developing a semantic layer. This paper describes the features provided by grid services [Foster et al 02; Tuecke et al 03] and describes their use to implement agent-based systems.

Our experience with multi-agent systems, and in particular with the Electric Elves project [Chalupsky et al 02], provided much of the motivation for this work. Some of the agents accessed travel web sites, others matched requests with agent capabilities, and others communicated with users about their travel needs. As we strived to use the system 24/7 inside and outside of our organization, we discovered that a non-trivial amount of effort was required to keep the groups of agents functioning. Multi-agent architectures and agent communication languages (such as OAA, ACL, KQML) address agent interaction issues in terms of their task-level behavior. However, many of the problems that we found are intrinsic to distributed computation, where heterogeneous collections of components need to be coordinated in a highly dynamic non-centralized execution environment. These are the environments that grid computing and grid services were designed to address.

The paper begins by motivating the need for a robust distributed infrastructure for agent systems, and lists six specific practical issues. We then describe briefly the features that grid services offer to address these issues. We then describe the use of grid services to implement an agent that will have robust continuous behavior. Finally, we discuss the advantages of the grid-based implementation and outline additional features provided by grid services that could also be exploited by agent systems.

Motivation

Figure 1 shows a representative implementation of an agent, using an example of airfare monitoring.

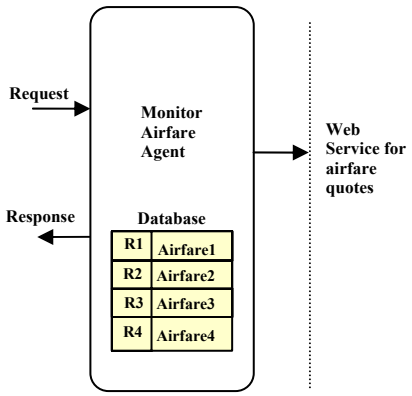

Fig. 1: A Typical Agent built from a Web service

This agent monitors airfare for a set of start time, origin airport and return time, destination airport, and notifies the user when the airfare changes.

The agent maintains a database of the requests that have come in so far. Each entry keeps information like the state

of the request (i.e. if the request is queued, being serviced, or has been serviced), the current airfare (or result) for this request, and so on. The requests are forwarded to an airlines web-service, in the appropriate format that the service uses, to get the latest airfare.

As the agents operate, it is necessary to ask various questions about their status and about the status of specific requests. In our experience, we found that we had to investigate by hand the answers to those questions, when clearly the agents should have been handling these themselves. Is the agent still working on my request? Does the agent still need my reply? Has the agent already responded but the response was not received? What is the status of my request? Other problems came up when machines went down. Should the requesting agents have a memory of their requests and resend them, or should the servicing agent keep track of their requests and reinstate them upon restart? Who should notice that an agent is not running and who should restart it?

The implementation of individual agents can be extended to address many of these issues. However, these problems clearly raise the need for better infrastructure to support robust continuous operation as tasks are executed in a dynamic distributed environment. Instead of developing ad-hoc solutions for these problems, it is useful to investigate whether we can build on existing approaches in distributed computing that address these issues at large.

The specific challenges can be summarized as follows:

1. **Introspection:** Once a request is sent to an agent, it would be useful to be able to query the agent regarding the receipt of the request, the status of the request, and perhaps any intermediate results that the agent may have generated. It would be useful to setup automatic notifications of changes in a request status.

2. **Tracking multiple requests:** For each agent, we have to implement a mechanism for tracking multiple requests. For each request, the agent has to maintain its status, current result, etc. This is done in the database shown in Figure 1. One issue we found is the overhead in detecting the same request only sent repeatedly when the message acknowledging its receipt was delayed.

3. **Persistence:** Each agent has to implement a database to store requests so they can be recovered in case the agent process dies. Special purpose scripts may be needed to check the agent process and restart it if it is not running.

4. **Scale:** Having a database brings up a problem of database storage size in case of large request traffic. Requests may be lost once the database is full. It would be useful to enable a requesting agent to inquire whether its request is still active, or better yet to guarantee that a request will be serviced regardless of the volume of requests from other sources.

5. **Shelf life of requests:** Each agent needs to implement a mechanism to deal with expiration of requests. Otherwise, all requests would be active forever. For example, monitoring a flight after the date of departure is not useful. Other services that may not have a clear date associated with them may be harder to handle in terms of terminating the request.

6. **Changes in specifications:** The agent specification (in many cases due to changes in the underlying web site) may change over time. This required manually tracking changes in agent specifications, so that request message formats could be updated.

Our work set out to investigate whether grid computing would provide adequate infrastructure to address these challenges.

Grid Services and the Open Grid Services Architecture

The Open Grid Services Architecture (OGSA) [Foster et al 02] supports the creation, maintenance, and application of collections of services across heterogeneous organizations that provide resources in a highly dynamic environment. This architecture builds on Web service standards and defines a set of extensions and conventions to support management of distributed services.

Grid Services are extensions of web services that incorporate two key features: soft state lifetime management and introspection through service data. OGSA supports the definition of grid service interfaces using the WSDL and XML Schema standards.

A key feature of grid services is soft state lifetime management, which is crucial for distributed computing. A request sent to a grid service may create a transient process called *grid service instance* with a unique identifier (a *grid service handle*) that can be used to locate and query the service.

A *Grid Factory* is used by a client/application to create a new grid service instance. The client invokes a CreateService operation on the factory, and receives a locator, or handle for the newly created service instance. This handle is then used to invoke operations upon the instance.

The grid factory creates a grid service instance and will maintain it for a limited amount of time, after which the instance will be destroyed. The expiration time of the instance can be set by the client/application. If required, the client/application can also extend the expiration time of the service instance. The grid factory can be designed to create service instances across pools of hosts with appropriate load balancing.

A second key feature of grid services is introspection. Grid services are stateful, which means that they maintain information across multiple operations. Therefore, they need a common mechanism to expose the data associated

with each service instance for query, update and change notification. This service instance state data is called *Service Data*. It describes the elements of a service that are externally observable, and can be queried or respond to change notification requests. This also supports third party notification of request status.

Each grid service instance can have *persistent properties* associated with it that are saved on permanent storage, and can be restored by the factory on a restart of the grid service instance in case the process is accidentally terminated (e.g., if the host goes down).

Grid services also adopt useful conventions for lifetime management. Grid services cannot use the same identifier if their interface specification is changed. This enables clients to have a handle on upgrades and changes to the spec through declarations of the shelf life of a service specification.

Note that some of the features of grid services are being considered for adoption by web service standards, such as service data declarations and access.

Implementing Agents as Grid Services

In this section, we describe the use of grid services for the agent architecture described earlier. The different components are shown in Figure 2, marked with labels (a) thorough (i) that we will refer to throughout the text. We provide details about the implementation and how to write similar agents as grid services in [Ratnakar 03].

The OGSA implementation in Globus Toolkit v.3 is used here which is available for download at http://www-unix.globus.org/toolkit/download.html.

Writing the Grid Service

We write an Interface (b) for the Grid Service, which includes an operation called monitorAirfare (e).

```
public interface Flight {
   public void monitorAirfare(String outdate, String indate, String
     fromAirport, String toAirport);
}
```

We also need to describe the "Service Data" of the grid service. Service Data is that information that the grid service will want to share with clients/applications that listen to notifications on that service data. The Service Data schema (c) for this service looks like:

```
<complexType name="MyAirFareQuoteType">
  <sequence>
    <element name="airlineName" type="string"/>
    <element name="fare" type="float"/>
    <element name="outwardTime" type="dateTime"/>
    <element name="returnTime" type="dateTime"/>
    <element name="errorOccurred" type="boolean"/>
    <element name="scrapeError" type="boolean"/>
    <element name="errorMessage" type="string"/>
  </sequence>
</complexType>
```

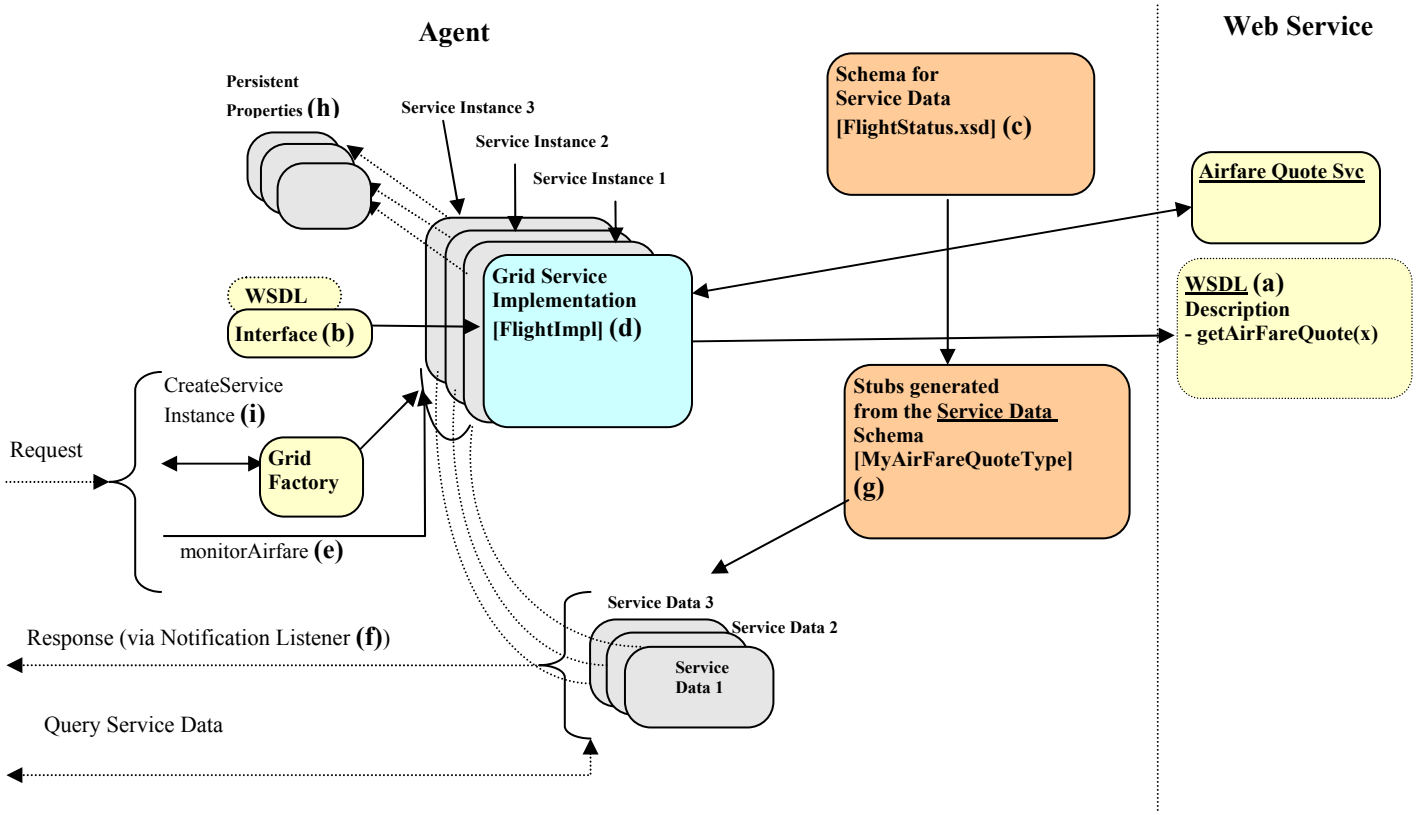

Fig. 2: Agent architecture using Grid Services

The last 3 elements in the service data (errorOccured, scrapeError and errorMessage) indicate errors in the SOAP call to the web service.

Java stubs/classes (g) are also generated from this description of the Service Data, so that we can refer to a "MyAirfareQuoteType" object inside the code.

Now, we finally start writing the implementation of the Flight interface (i.e. body of the monitorAirfare function above), called FlightImpl (d). The FlightImpl class is created by extending the Notification Service Skeleton class, thereby making this Grid Service a notification service (i.e. which can send out notifications on a change of its service data to all the clients that have subscribed to that service data). Depending on the type of service to be created, we would extend the appropriate Service Skeleton class while defining the implementation class.

When we get the request data through the monitorAirfare call, we invoke the appropriate operation on the AirfareQuote web service. Once we get the results back, they are sorted and the cheapest flight result is checked. If this has changed from the earlier value, then the service data elements are changed accordingly and a notification is sent out. The service instance continuously keeps polling for lower fares like this until the instance expires (either through an explicit expiry time set during the time of creation of the instance, or via an argument passed to monitorAirfare).

Also, whenever the request to monitorAirfare comes in, we can store the request as "persistent properties" (h) (a feature inbuilt into grid services).

```
setPersistentProperty("indate", indate);
```

We also restore the persistent properties (if present) in the "PostCreate" function (the function which gets called whenever a service instance is created).

```
String indate=(String)getPersistentProperty("indate");
```

We can make the service instance also persistent, by indicating so in the service deployment descriptor file:

```
<parameter name="instanceLifecycle" value="persistent"/>
```

Thus, if the server crashes, and restarts, the instance is created again, the persistent properties are read and the request data (or any other data that needs to be saved/restored) is restored.

Writing the Agent

The agent is composed of two parts. One is used to actually invoke the monitorAirfare method (e), and the other is used to listen for notifications (f) on the 'MyAirFareQuoteType' service data.

First we need to make a call, CreateService (i), to the Grid Factory Service which actually creates an instance of the Flight service, and returns a handle to that instance. This handle is referred to in all communications by the client (i.e. monitorAirfare, and listening for notifications). Note that each service instance has its own set of persistent properties (for recovery), and its own service data (for notifications).

Invoking the Service

To invoke the service, we first create an instance of the service, and then use the handle returned in the first step to make further calls. These steps show example interactions with the flight monitoring agent as a grid service:

- First, start a grid service container (i.e. the Grid Hosting Environment) by typing "ant startContainer" in the main grid toolkit directory.

- Create a new service instance:
  % java org.globus.ogsa.impl.core.factory.client.CreateService http://localhost:8080/ogsa/services/travel/FlightMonitorService myflight

- In another window, start the notification listener :
  % java org.globus.ogsa.travel.impl.FlightStatusListener http://localhost:8080/ogsa/services/travel/FlightMonitorService/myflight

  >> Waits for responses.. example response :
  Price : $405.3
  Airlines : Travelocity
  Leaving : 2003-4-28, 9:40:0 AM
  Returning : 2003-5-9, 7:0:0 AM

- Give a request to monitor the airfare for a certain date and between certain airports :
  % java org.globus.ogsa.travel.impl.FlightClient http://localhost:8080/ogsa/services/travel/FlightMonitorService/myflight monitorAirfare "Apr 26, 2003" "Apr 29, 2003" "LAX" "JFK"

Challenges Revisited

The challenges raised earlier are addressed by the grid services framework as follows:

1. Introspection:
The grid service data provides the infrastructure needed to track requests and to enable clients to setup notifications for updates to the status of the request.

2. Tracking multiple requests:
The agent had an overhead of dealing with multiple requests. Here, we don't have to worry about multiple requests in the implementation of the service (FlightImpl), since each request spawns off a new service instance with its own properties and service data.

3. Persistence:
This is addressed by the "persistent properties" of grid services. We do not lose data on a machine crash now

because persistent properties are being used.

4. **Scale**:
   The database overflow problem, or the problem of having too many requests is partially solved by the above mechanism (i.e. expiring old requests automatically), but it doesn't work for bursts of request activity. Since each request essentially starts up a new process (the service instance), we can create an intermediate "Broker" service, which intercepts the call to "CreateService", and can decide to create the instance on any machine from a pool of hosts.

5. **Shelf life of requests**:
   There is no standard mechanism to set the expiration of the request for an agent system. In the grid services infrastructure, we can explicitly set the lifetime of the service instance at the time of creation (CreateNewService). Also, all service instances are destroyed after their expiration time, so there is an explicit way to handle obsolete requests.

6. **Changes in specifications**:
   Services have a unique identifier that remains the same as long as they have the same interface specifications. If those specifications change, the service will have a new identifier. The specifications may have an expiration date indicated as a guarantee of validity. This does not solve the problem, but at least the client/application has an explicit way to be informed about it.

Related Work

There are a few languages to express service compositions that build on WSDL. These include the WFDL Web Services Flow Language [Christensen et al 01] and XLANG [Thatte 01]. A related aspect is the sequencing of message exchanges supported by a service, as in the Web Services Conversation Language (WSCL) [Banerji et al 02]. Other related standards are BPMI's Business Process Modeling Language (BPML), ebXML's Business Process Specification Schema (BPSS), and NIST's Process Specification Language (PSL). DAML-S is a semantic markup language for services that enables the expression of a complex service as well as the composition of services [Ankolenkar et al 02]. All of them are currently under development. We will consider the suitability of any emerging standard as we proceed with our work.

Some languages to support composition of services are based on expressive formalisms to represent complex combinations of services [McIlraith and Fadel 02; McIlraith and Son 02]. These languages include, for example, conditional expressions. This work is complementary in that it investigates the logical underpinnings of such languages, while our focus is on the usability of the language.

Conclusions

Grid services offer an infrastructure for distributed computing that supports robust coordination in highly dynamic environments. Semantic web services can be built as extensions to grid services and take advantage of their mechanisms to address introspection, lifecycle management, and persistence. Service data can also be exploited to create semantic descriptions of request execution status.

In addition to the mechanisms discussed in this paper, grid computing can also provide a well-developed infrastructure to address including security, cross-organization access policies, load balancing, efficient data transfer and execution. Execution management in dynamic environments is a key issue for multi-agent systems, and semantic web services should be built on paradigms that support these distributed computing issues. Grid services provide key technology for robust distributed problem solving.

Acknowledgements

We would like to thank Carl Keseselman and other members of ISI's Center for Grid Technologies for their technical support with OGSA and their comments on this work, including Ben Clifford, Carl Czajkowski, Ewa Dweelman, and Hongsuda Tangmunarunkit. Thanks also to Jim Blythe, Jihie Kim, Craig Knoblock, and to other researchers involved in Electric Elves. This research was supported in part by the National Science Foundation under grant EAR-0122464 (SCEC/ITR), and in part by an internal grant from USC's Information Sciences Institute.

References

[Ankolenkar et al 02] A. Ankolenkar, M. Burstein, J. R. Hobbs, O. Lassila, D. L. Martin, D. McDermott, S. A. McIlraith, S. Narayanan, M. Paolucci, T. R. Payne, and K. Sycara. *DAML-S: Semantic Markup for Web Services*. In The First International Semantic Web Conference (ISWC), Sardinia (Italy), 2002.

[Banerji et al. 02] Banerji, A. et al. *WSCL: Web Services Conversation Language.* W3C Note, March 2002.

[Chalupsky et al. 02] Hans Chalupsky, Yolanda Gil, Craig Knoblock, Kristina Lerman, Jean Oh, David Pynadath, Tom Russ, Milind Tambe. *Agent Technology to support Human organizations*. In AI Magazine, Vol 23, No 2, Summer 2002.

[Christensen et al 01] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *WSDL: Web Service Description Language*, http://www.w3.org/TR/wsdl, 2001.

[Foster et al 02] Foster, I., Kesselman, C., Nick, J. and Tuecke, S. *The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems*

Integration. Globus Project, 2002, www.globus.org/research/papers/**ogsa**.pdf

[McDermott 02] McDermott, D. *Estimated-Regression Planning for Interactions with Web Services*. in Sixth International Conference in AI Planning Systems, 2002.

[McIlraith and Fadel 02] McIlraith, S. and Fadel, R. *Planning with Complex Actions*. Proceedings of the Ninth International Workshop on Non-Monotonic Reasoning (NMR2002), pages 356-364, April, 2002.

[McIlraith and Son 02] McIlraith, S. and Son, T. Adapting *Golog for Composition of Semantic Web Services*. Proceedings of the Eighth International Conference on Knowledge Representation and Reasoning (KR2002), Toulouse, France, April, 2002.

[Narayanan and McIlraith 02] Narayanan, Srini, and McIlraith, Sheila A. *Simulation, Verification, and Automated Composition of Web Services*. Proceedings of the World Wide Web Conference, Honolulu, Hawaii, 2002.

[Ratnakar 03] Ratnakar, V. *Writing an airfare quote grid service*. http://www.isi.edu/ikcap/cognitive-grids/airfare/grid-3.doc 2003

[Thatte 01] Thatte, Satish. *XLANG: Web Services for Business Process Design*. Microsoft Report, 2001.

[Tuecke et al. 03] Tuecke, S. et al. The Open Grid Services Infrastructure (OGSI), V 1.0, April 2003.