Message Tracking in SOAP-based Web Services

A. Sahai, V. Machiraju, J. Ouyang, and K. Wurster
Hewlett-Packard Laboratories
1501 Page Mill Rd MS 1U-14
Palo Alto, CA 94304
USA
{asahai,vijaym}@hpl.hp.com

Abstract

As web services become more prevalent, the nature of electronic transactions on the Internet changes from simple browser-to-business clicks to an orchestrated flow of messages between cross-enterprise services. Consequently, more than one service could participate in the federated execution of a single transaction. In such cases, the problem of end-to-end management becomes very important. The inherent cross-enterprise or distributed nature of the problem, security of information exchanged, and the complexity in correlating related messages into a single transaction make this problem challenging. In this paper, we present an approach to track and correlate messages between web services that are part of a single transaction. We do this by proposing management information exchange agreements between service providers, and a distributed message tracking algorithm that is executed within each service provider. We also explain the techniques for realizing our solution in the case of web services that communicate using the Simple Object Access Protocol (SOAP).

Keywords

End-to-end management, web services, message tracking, cross-enterprise management, SOAP.

1. Introduction

Web services can be described broadly as applications capable of executing transactions[1] via the Internet. One refers to a web service on the Internet through a Uniform Resource Locator (URL) and invoke its transactions by sending messages in a vocabulary it understands. Web services are set up to be accessed either by clients (e.g., browsers, appliances) or by other web services. Enabling interoperation between these services requires agreement on several standards. UDDI (Universal Descrip-

1. We use the word transaction in a weak sense to refer to portions of business logic that are executed by a service. This is in contrast to the usage of this term in the database literature.

tion, Discovery, and Integration) [1] is an emerging standard for web services to register and make themselves available to clients or other web services. WSDL (Web Services Description Language) [2] is another standard for web services to describe their capabilities or transactions. *Simple Object Access Protocol (SOAP)* [3] defines a base communication protocol for clients and services to exchange XML messages with each other. Several other domain-specific standards (e.g., ebXML [4]) are emerging to define semantics of messages exchanged between web services.

SOAP is a protocol for exchange of information in a decentralized environment such as the Internet. It is an XML based protocol that defines how messages should be created and exchanged between web services. More precisely, SOAP consists of three parts: an envelope that defines a framework for describing what is in a message and how to process it, a set of encoding rules for expressing instances of application-defined data types, and a convention for representing remote procedure calls and responses. SOAP can work over a variety of communication level protocols, most implementations support at least HTTP.

Transactions in web services are federated across enterprises, and SOAP enables such federation by defining a protocol for exchange of messages between web services. A *transaction* is a portion of business logic with a clearly defined begin-point and end-point. As part of executing a transaction, a web service can send one or more SOAP messages to other web services and/or receive one or more SOAP messages. The problem of *message tracking* is to be able to track all the messages that are part of one such transaction through all the web services the transaction executes in. Message tracking helps in end-to-end management of web services since one can infer metrics for the whole transaction and for portions of the whole transaction from messages that flow between web services.

In this paper, we describe an infrastructure for tracking messages that enables end-to-end management in federated web services. We assume that these web services interact using SOAP or other similar protocols. The rest of this paper is organized as follows: In section 2, we provide examples of web services and SOAP messages, and explain the problem of end-to-end management in detail. Section 3 describes our approach to solving the problem. In section 4, we describe typical implementations of our approach, and we summarize our conclusions and directions for future work in section 5.

2. Background and Problem Definition

Web services are federated in nature as they interact across management domains and enterprise networks. Their implementations can be vastly different in nature. Some of the common technologies for implementing web services are J2EE[1], SUNONE[2], and .Net[3]. Figure 1 shows a fictional example of interacting web services. Employees in a

1. Java 2 Enterprise Edition from Sun Microsystems (http://java.sun.com/j2ee)
2. Sun Open Net Environment from Sun Microsystems (http://www.sun.com/sunone)

company (workhard.com) use their internal web service (supplies.workhard.com) to order day-to-day supplies and stationery. The internal supplies web service in turn uses a supplies marketplace (supplies.marketplace.com) to find the best deals. The marketplace requests bids from two supplies companies (stationery.com and office-supplies.com) to fulfill orders. Both the suppliers use a shipping service (shipme.com) for shipping the ordered products directly to the consumer.

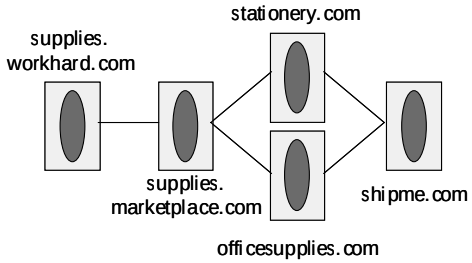
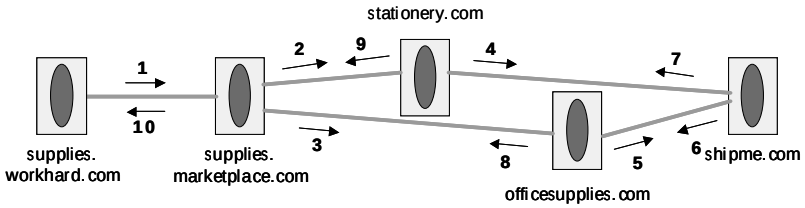


Figure 1: Interacting web services

When two web services connect to each other, they have to agree on a document exchange protocol and the appropriate document formats. From then on they can interoperate with each other exchanging documents. SOAP defines a common layer for document exchange. Services can define their own service-specific content on the top of SOAP. The execution of a single business transaction can involve multiple messages being exchanged between web services. For example, a purchase order transaction that begins when an employee orders supplies and ends when he or she receives a confirmation could result in ten messages being exchanged between various services as shown in Figure 2.



1. purchase order
2. part of the purchase order
3. the other part of the purchase order
4. shipping request
5. shipping request
6. shipping confirmation
7. shipping confirmation
8. order confirmation
9. order confirmation
10. purchase order confirmation

Figure 2: SOAP messages exchanged between web services

---

3. .Net from Microsoft (http://www.microsoft.com/net/default.asp)

One such message encapsulated in SOAP is shown in Figure 3. Note that every SOAP message has a clearly defined header (SOAP-ENV:Header) and body (SOAP-ENV:Body). The service-specified content is enclosed within the body of a SOAP message. Headers are typically used to represent meta-information about messages (e.g., message identifier).

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header>
    <Id>1</Id>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <PurchaseOrder>
      <Item count = 100>Postit stick notes</Item>
      <Item count = 200>Stapler</Item>
    </PurchaseOrder>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 3: SOAP message

The exchange of messages between web services could be asynchronous. Services sending a request message need not be blocked waiting for a response message. In some cases, all the participating services are like peers, in which case there is no notion of a request or a response. Some of the message flow patterns that result from this asynchrony are shown in Figure 4. The first example in Figure 4 shows a single request resulting in multiple responses. The second example shows a broker-scenario, in which a request is sent to a broker but responses are received directly from a set of suppliers.

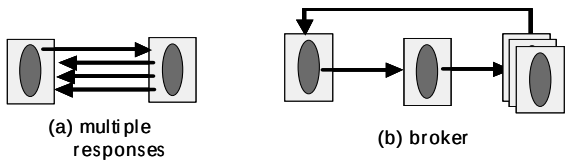


Figure 4: Asynchronous message patterns between web services

Management of web services is a challenging task because of their heterogeneity, asynchrony, and federation [5, 6]. Motivations for end-to-end web service management arise from the perspective of both clients and service providers. Clients are interested in tracking their requests and in understanding bottlenecks or causes for failure of their requests. Measurements taken at various points along the execution of a transaction are very helpful in analyzing end-to-end quality of the transaction. Service providers that are using other services to provide composite services would like to know how the component services are behaving. By studying and observing their behavior a composite web service would be able to optimize itself by either changing its component sub services or by instructing the existing component sub services to improve performance.

Message tracking is an enabler for end-to-end transaction management. It enables clients and web services to track the flow of messages related to a single transaction between web services. More formally, using the terminology from graph theory, the problem of message tracking is to be able to build a *forest of trees* where:

- Each tree in the forest represents a transaction.
- Each node in a tree represents a message exchanged between web services as part of executing that transaction.
- A parent-child relationship between two nodes indicates that the message corresponding to the child node was exchanged in the context of the message corresponding to the parent node.
- Each node can have a number of associated management attributes.

Figure 5 shows an example of a tree corresponding to the exchange of messages from Figure 2.

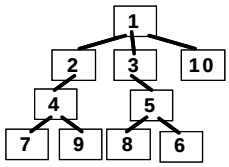


Figure 5: A tree of messages that are part of a single transaction

There are various constraints for solving this problem. Some of these arise by the nature of the web services interactions. Others are desirable for a scalable solution involving a large number of interacting web services.

- Since web services are autonomous, the message tracking process should also be executed in a decentralized way and should not assume the existence of a centralized consolidator.
- If the message tracking process requires exchange of additional information between the participating web services, such information should be captured and agreed upon between the services through explicit agreements.
- The message tracking process should be able to handle asynchronous communication of messages between web services.

3. Message Tracking

Our approach to message tracking hinges on two concepts that we have defined: message detail records (MDRs) and agreements between service providers. These two concepts are explained next.

3.1 Message Detail Records (MDRs)

A message detail record is a collection of management attributes that is padded to every message flowing between web services. The structure of an MDR depends on the management information that is agreed to be exchanged between two parties (see next section for a complete discussion on agreements). The sender of the message

could fill in some of the attributes in an MDR, while the receiver fills in the rest. For example, a simple MDR structure could consist of a unique identifier for the message, a time stamp when the message was sent, and a time stamp when the message was received. In this example, the sender should fill in the first two attributes and the receiver fills in the last one.

When a new message is sent out by a service, this message is either the first message originating a new transaction, or is a message that is being sent out in the context of another earlier message, which already originated a transaction. In the first case, the MDR of the new message does not have any relationship with other MDRs. In the second case, however, the MDR of the new message is related to the MDR of the earlier message through a parent-child relationship. This relationship captures the ongoing trace of distinct messages being exchanged as part of a single transaction. A complete definition of a typical MDR that captures the parent-child relationships is shown in Figure 6.

```
MDR
{
   parent_mdr   : message detail record of the parent message
   message_id   : unique identifier of the message
   message_type : type of the message
   source       : identifier of the service originating the message
   target       : identifier of the service receiving the message
   time_sent    : time when the message was sent by source
   time_recd    : time when the message was received by target
}
```

Figure 6: Message detail record (MDR)

In the case of SOAP messages, the SOAP header is a convenient place to insert an MDR. A sample SOAP message with an MDR is shown in Figure 7.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
 <SOAP-ENV:Header>
 <MDR>
      <parent_mdr></parent_mdr>
      <message_id>1122-3551-5721-8834</message_id>
      <message_type>PurchaseOrder</message_type>
      <source>supplies.workhard.com</source>
      <target>supplies.marketplace.com</target>
      <time_sent>2001-06-12 12:00:23</time_sent>
      <time_received></time_received>
 </MDR>
 </SOAP-ENV:Header>
 <SOAP-ENV:Body>
   <PurchaseOrder>
      <Item count = 100>Postit stick notes</Item>
      <Item count = 200>Stapler</Item
   </PurchaseOrder>
 </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 7: MDR in a SOAP message

3.2 Agreements

The nature of the management information exchanged between web services and the security of such information is captured in agreements. An agreement is a description of:

- The list of attributes that are part of an MDR
- The rules for propagating those attributes to other web services.

An agreement is always formed between two web services. All the web services along the chain of a business transaction should have separate pair-wise agreements between each other. This is usually the case in traditional service-level-agreements (SLAs) where services agree pair-wise on the terms of their relationship. While SLAs capture the terms regarding the execution of each other's transactions and the associated quality parameters, the agreements we refer to here are about the exchange of special management information. A possible implementation of our agreements is to include them as additional clauses in traditional SLAs.

Why should web services exchange additional management information? The reason is for them to collaborate in order to do end-to-end management. No single web service can create a complete picture of the end-to-end transaction. Exchanging management information allows each of the participating web services to define and measure end-to-end metrics, and maybe even enforce certain end-to-end goals, which are not captured in individual SLAs.

The additional management information in our approach is described through MDRs. One set of clauses in the agreement is regarding the structure of an MDR. For example, two services may agree upon exchanging time stamp information with each other. Two other services may agree upon exchanging QoS expectations as part of an MDR. In the worst case, the two services may agree to exchange no MDRs in their messages.

A second set of clauses is needed around the security of MDRs. We have already discussed how a service can propagate an MDR that it received as a parent of a new MDR that it created. Consider a service A that sends a message X to service B. In the course of handling message X, service B creates a new message Y to be sent to service C. To create an MDR for Y, service B has the option of (Figure 8):

1. Just including the MDR of Y in message Y.
2. Including the MDR of Y, but also including the MDR of X as the parent of Y's MDR.

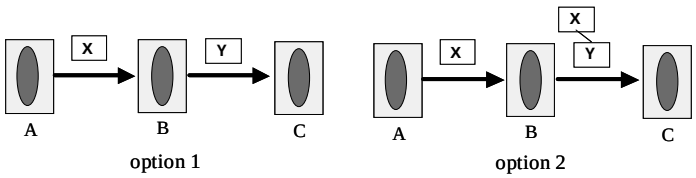


Figure 8: Propagation of MDRs

In the second case, some of the attributes of the message exchanged between A and B are shared with C. The security of such information depends on the exact nature of the attributes. For example, if the attribute is an opaque identifier that does not reveal the identity or any other characteristic of A, then A may allow it to be propagated to C. In either case, whether an attribute of an MDR can be propagated by the receiver to its component services has to be explicitly stated as part of the agreement. An example of an agreement between two web services is shown in Figure 9.

```
Agreement
{
   parent_mdr   : propagate
   message_id   : propagate
   message_type : do not propagate
   source       : do not propagate
   target       : propagate
   time_sent    : propagate
   time_recd    : propagate
}
```

Figure 9: Agreements

3.3 Distributed Message Tracking Algorithm

We are now ready to present a message-tracking algorithm that executes decentrally within each service provider. The algorithm has two parts - a part that executes whenever a service sends out a message, and a part that executes whenever a service receives a message. As the algorithm is executed, a data structure call MDRForest gets built, which corresponds to the forest of trees described in section 2. The MDRForest is empty when the algorithm starts execution.

The portion that executes before a message is sent out is shown in Figure 10. To summarize, a new MDR is created and stored with the right parent-child relationships in the MDRForest. Then, an appropriate sub-tree of the tree containing the new MDR in MDRForest is inserted into the message. This requires trimming the tree (removing certain attributes or removing nodes from the tree) to ensure that all agreements as described in section 3.2 are satisfied.

```
Before sending a message M in the context of message C
{
   1. Create new unique identifier (new_id).
   2. Create a new mdr and add identifier from step 1 to mdr
      new_mdr.id <- new_id
   3. Fill up other fields in new_mdr
   4. Search for C's mdr in MDRForest (c_mdr)
   5. add new_mdr as the child of c_mdr in MDRForest
      c_mdr.children <- new_mdr
   6. Get agreement-compliant sub-tree of new_mdr from MDRForest
      (mdr_sub_tree)
   7. Set mdr_tree in message header.
      M.header.mdr <- mdr_sub_tree
}
```

Figure 10: Message tracking algorithm for outgoing messages

The portion that executes whenever a message is received is shown in Figure 11. The first step is to extract the MDR from the incoming message and fill up any fields in that MDR. The second step is to merge the MDR with the MDRForest. Merging involves replacing any of the existing nodes in the MDRForest with new nodes from the incoming MDR, if they represent the same message. This ensures that the information in MDRForest is always up-to-date.

```
After receiving a message M
{
   1. Extract mdr_tree from M.header
   2. Merge mdr_tree with MDRForest
      MDRForest.merge <- mdr_tree
}
```

Figure 11: Message tracking algorithm for incoming messages

As the algorithm is executed at each web service, the MDRForest that gets built up gets richer and richer with every incoming or outgoing message. Agreements permitting, every web service will be able to see the entire tree of messages along a transaction chain starting with the first message that started the transaction, and ending with the last message that is sent out by that web service as part of that transaction. Figure 12 shows a trace of this algorithm as it executes for the messages from Figure 2.

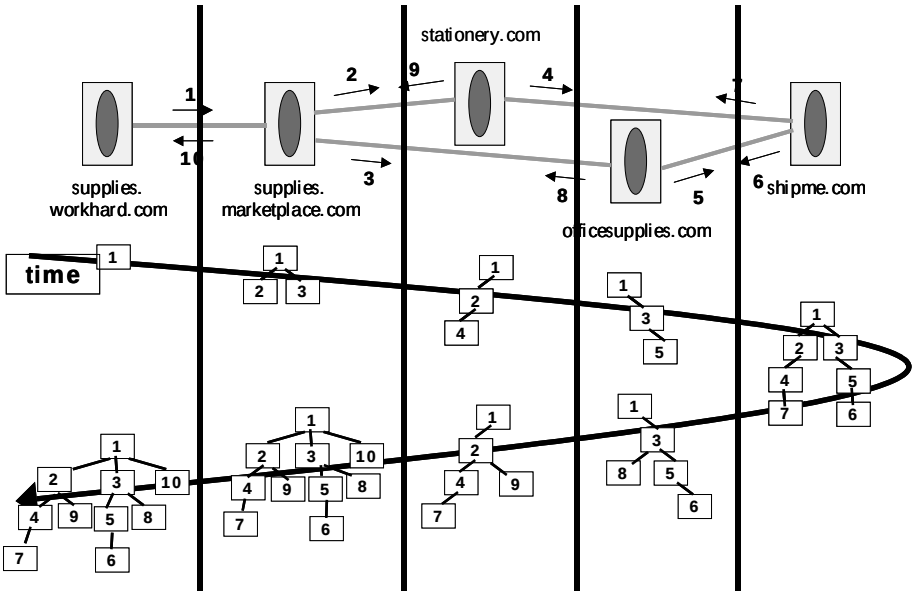


Figure 12: A trace of message tracking algorithm

The vertical lines in the above picture show the enterprise boundaries; the trees that get built up as part of the MDRForest within each web service are shown under

the corresponding web service. The second set of trees within each web service get merged with the first set of trees, thereby resulting in only one tree for this transaction at every web service.

The trees shown in Figure 12 are a result of all the web services agreeing to propagate MDRs to each other. If one of the web services along the chain of transaction does not propagate an MDR or propagates only portions of an MDR, then subtrees of those shown in Figure 12 would result.

There are issues in the message-tracking algorithm that need further clarification: How does one specify the contextual relationship between two messages? Can the relationship be automatically inferred by the algorithm or should it be specified by the web service developer? These issues are addressed in the next section, where we describe our demonstrative implementation of the message-tracking algorithm over SOAP infrastructure.

4. Implementation

We implemented the web services shown in Figure 1 using a standard J2EE-compliant application server and web server. Each of the web services was implemented as a servlet capable of receiving and sending messages. The messages were themselves implemented as SOAP messages, and for this purpose we used the Apache SOAP toolkit[1]. HTTP protocol was used as the transport for SOAP messages. Using standard Internet protocols such as HTTP and infrastructure such as web servers helped us cross firewall boundaries between web services while leveraging existing security mechanisms. Higher levels of security (e.g., application-level security) can be achieved by adding additional fields in SOAP headers. The issue of time synchronization between web services was not handled as there are existing protocols to provide time synchronization in such cases [7].

The message-tracking algorithm was implemented as an agent. Web services used the agent's API to inform it of incoming or outgoing messages. We first describe this API-based approach in the next sub-section. We also describe an interception-based approach at the end of this section.

4.1 Message Tracking API and Agent

Based on the data structures and distributed correlation algorithms presented above, an API to interact with the message-tracking agent has been developed. The agent can be used to monitor message flows, measure, and break down the response times of web service transactions. The agent also has a user interface to visualize the messages and transactions. The API is defined as follows.

```
public class Mtrack
{
   public static void init();
   public static String sendMessage(String xmlMsg,
```

1. Available for download at http://xml.apache.org/soap/

```
                                    String parentMsgId,
                                    String msgType,
                                    String srcService,
                                    String desService);
   public static String recvMessage(String xmlMsg);
}
```

Figure 13: Message tracking agent API

When a web service gets started, it calls Mtrack.init() to initialize the message tracking agent. Before sending a message, the web service calls Mtrack.sendMessage() by passing the message to be sent, the parent message ID, the message type, and the identifiers of the sending and receiving services. The agent constructs the MDR as per the algorithm described above, attaches the MDR to the message, and returns the modified message. After receiving a message, the web service needs to call Mtrack.recvMessage() by passing the received message.

The agent builds the MDRForest and a collection of aggregate metrics such as average transfer times, average processing times, average response times, and counters on number of messages received or sent. The transfer time for any message is the duration between the times the message is sent by one service and received by the other. The average transfer time for a type of message is the sum of transfer times for the message instances of this type, divided by the number of such messages recorded. The processing time at a service is defined for every incoming message as the duration between the time it was received and the time when the next message is sent out in the context of the incoming message. The response time at a service is defined for every outgoing message as the duration between the time when a request is sent out and the time when the response is received.

Figure 14 and Figure 15 show snapshots from the user interface of the message tracking agent in workhard.com. The first graph shows aggregate metrics for a particular type of business transaction - average transfer times and the number of messages for each type of message. The second graph shows a drill-down view into an instance of a business transaction at workhard.com. On this graph, one can see the exact time stamps at each of the web services the transaction executed through. Figure 16 is a visualization of the same type of information collected by the message tracking agent at officesupplies.com. One can notice the difference in the end-to-end views as observed from different services along the business transaction.

4.2 Message tracking by interception

The implementation approach described in the previous section requires the web service developer to use the message tracking API and insert calls to this API in the code for the web service. The task of relating one message to the next is faciliated by the API described in Figure 13 (*parentMsgId* argument in *sendMessage* method). In that approach, a web service developer who understands the business logic behind messages sent and received will be able to use the API calls to establish the relationships between incoming and outgoing messages. There is an alternate approach that is non-

intrusive in nature to the web service. This approach is called message tracking by interception and is described below.

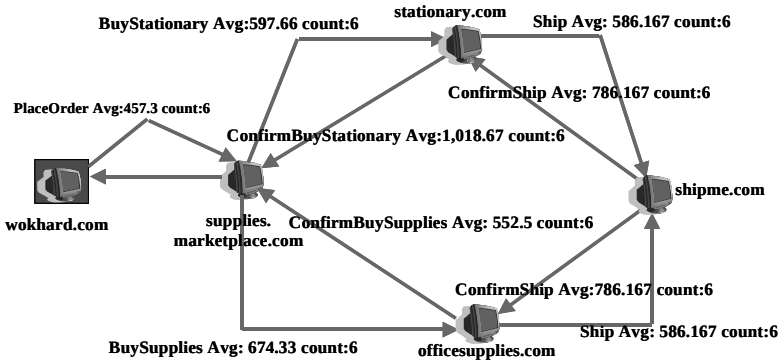


Figure 14: Aggregate view from the agent in workhard.com

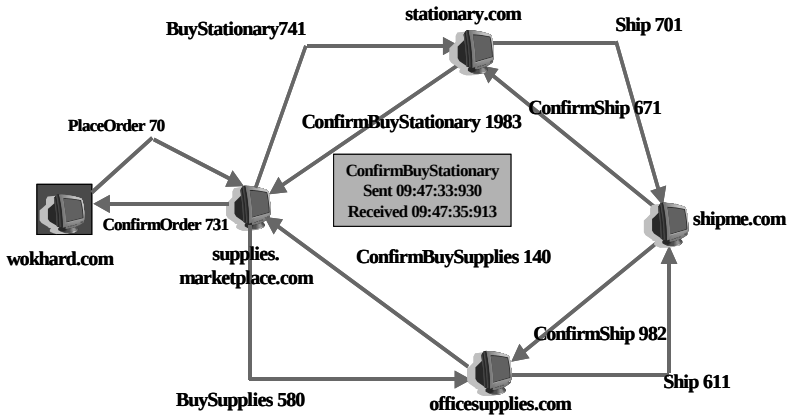


Figure 15: Transaction view from the agent in workhard.com

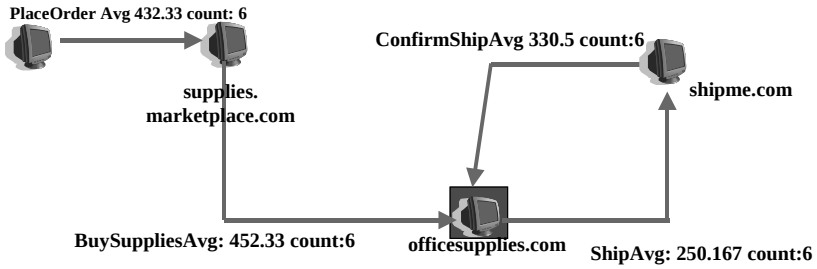


Figure 16: Aggregate view from the agent in officesupplies.com

Most of the web services are built on middleware platforms. These platforms provide communication services among others. A SOAP communication service, for example, helps in creating, sending, receiving, and parsing SOAP messages. Such a communication service could be instrumented to execute special logic whenever a message is sent or received. Most of the middleware platforms provide hooks (e.g., ISAPI/NSAPI[1] filters in web servers, servlet redirection in application servers, hooks for inserting plug-ins in SOAP engines, etc.) for adding such instrumentation [8, 9]. One can create instrumentation modules to automatically call the message tracking agent described in section 4.1. Besides freeing service developers from having to instrument their web services, this approach also allows legacy web services to take advantage of message tracking.

However, there is one drawback with this approach. The middleware-based approach will not be able to correlate one message with another. In other words, it will not be able to tell if a message is sent in the context of another. For this to be done automatically, we need a higher level of middleware - a middleware platform that understands not only the template for messages, but also the types of messages, and the flow pattern of messages. For example, a conversation engine or a process-flow engine [10, 11] would be able to correlate messages by putting one in the context of another.

5. Conclusion

Tracking messages that are part of a business transaction as the transaction flows through multiple web services is an important enabler for end-to-end management. We have proposed a message tracking algorithm that executes within each service provider in a decentralized manner. This algorithm does not require any additional messages between web services to perform such tracking. Instead, it pads messages from ongoing interactions with extra management information. The extra management information is represented in a data structure called message detail record (MDR) and can easily be inserted into the header of a SOAP message. Further, the exact information that is put into an MDR is flexible, thereby allowing different forms of end-to-end management. For example, if one uses MDRs to insert message time stamp data (the times when a message is sent or received), one can perform end-to-end transaction performance management. Similarly, if one inserts source and target service identifiers into an MDR, one will be able create end-to-end service topology views.

The security around propagating management information is governed by pair-wise agreements between web services. These agreements form the basis for how much information can be propagated from one service to another by a service that resides between the two. Such agreements can be pre-negotiated between the partici-

1. ISAPI (Internet Server API) and NSAPI (Netscape Server API) are programming APIs for Microsoft Internet Information Server and Netscape Server respectively.

pants ahead of time (for all the transactions over a period of time) or can perhaps be automatically negotiated on a transaction-by-transaction basis in the future.

As more advanced forms of middleware platforms become available, one will be able to perform message tracking and end-to-end management in a manner that is non-intrusive to web service developers. Research into such forms of middleware and intelligent analysis of collected end-to-end data are topics for further research.

References

[1]. Universal Description, Discovery, and Integration, http://www.uddi.org.

[2]. Web Services Description Language, http://www.w3.org/TR/wsdl.

[3]. Simple Object Access Protocol, http://www.w3.org/TR/soap.

[4]. E-business XML, http://www.ebxml.org.

[5]. Brown, A., Kar, G., Keller, A., An Architecture for Managing Application Services over Global Networks. Proceedings of the 7th International IFIP/IEEE Symposium on Integrated Management (IM 2001), IEEE Press, May, 2001

[6]. Ensel, C., Keller, A., Managing Application Service Dependencies with XML and Resource Description Framework., 7th International IFIP/IEEE Symposium on Integrated Management (IM 2001), Seattle, WA, USA, May, 2001

[7]. D. Mills. Simple Network Time Protocol. RFC 2030, 1996

[8]. Gschwind, Thomas; Eshghi, Kave; Garg, Pankaj K.; Wurster, Klaus. Web Transaction Monitoring. HPL-2001-62, http://www.hpl.hp.com/techreports/2001/HPL-2001-62.html

[9]. Frolund, Svend; Pedone, Fernando; Pruyne, Jim; van Moorsel, Aad. Building Dependable Internet Services with E-speak. HPL-2000-78.http://www.hpl.hp.com/techreports/2001/HPL-2000-78.html

[10]. Harumi Kuno, Mike Lemon, Alan Karp. Transformational Interactions for P2P E-Commerce. Thirty-fifth Annual Hawai'i International Conference on System Sciences (HICSS 2002). Also HPL-2001-143, http://www.hpl.hp.com/techreports/2001/HPL-2001-143.html

[11]. Web Services Flow Language, http://www.ibm.com/software/solutions/webservices.