HEWLETT PACKARD

End-to-End E-service Transaction and Conversation Management through Distributed Correlation

Akhil Sahai[1], Jinsong Ouyang[2], Vijay Machiraju[1], Klaus Wurster[2]
Software Technology Laboratory
HP Laboratories Palo Alto
HPL-2000-145
November 7th, 2000*

end to end, transaction, conversation, E-service, management, Web, Internet

With the widespread deployment of Internet, E-services are becoming prevalent. E-services are being created in the form of portals and e-business sites. They interact amongst themselves to provide a range of functionality to clients. E-services thus undertake static composition. There is an increasing trend towards dynamic composition, where e-services choose dynamically their trading partners. As these e-services are being deployed by different enterprises, they are distributed and federated in nature. They also have varied implementations. In addition, these e-services undertake conversations that involve multiple interactions between e-services, which is asynchronous and often asymmetric in nature. End-to-End management of e-service conversations and thereby their transactions is therefore a challenging task. A distributed correlation approach is presented that enables end to end correlation of conversations and transactions spanning multiple e-services in a distributed and decentralized manner. The distributed correlation mechanism obviates the need of a central correlation engine as the correlation data is sent along with the documents exchanged amongst the e-services.

* Internal Accession Date Only
Approved for External Publication
[1] HP Laboratories Palo Alto
[2] HP Open View Business Unit
© Copyright Hewlett-Packard Company 2000

End-to-End E-service Transaction and Conversation Management through Distributed Correlation

Akhil Sahai[1], Jinsong Ouyang[2], Vijay Machiraju[1], Klaus Wurster[2]

E-services Solutions and Management Department , HP Laboratories
1501 Page Mill Road, Palo-Alto, CA 94034

Abstract: With the widespread deployment of Internet, E-services are becoming prevalent. E-services are being created in the form of portals and e-business sites. They interact amongst themselves to provide a range of functionality to clients. E-services thus undertake static composition. There is an increasing trend towards dynamic composition, where e-services choose dynamically their trading partners. As these e-services are being deployed by different enterprises, they are distributed and federated in nature. They also have varied implementations. In addition, these e-services undertake conversations that involve multiple interactions between e-services, which is asynchronous and often asymmetric in nature. End-to-End management of e-service conversations and thereby their transactions is a therefore a challenging task. A distributed correlation approach is presented that enables end to end correlation of conversations and transactions spanning multiple e-services in a distributed and decentralized manner. The distributed correlation mechanism obviates the need of a central correlation engine as the correlation data is sent along with the documents exchanged amongst the e-services.

A. INTRODUCTION

An *e-service* is a service available via the Internet that completes tasks, solves problems, or conducts transactions. These e-services are accessible on the Internet at a particular Uniform Resource Locator. An e-service may depend on other e-services. These e-services are termed composite e-services. This composition could be static or dynamic in nature.

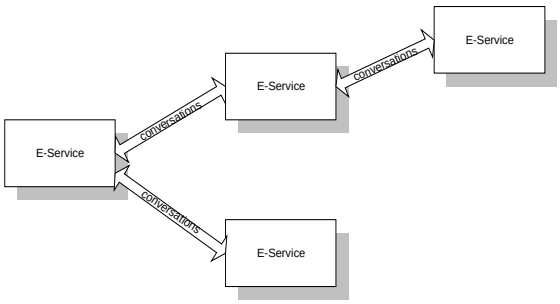

Figure 1. E-services undertake conversations

These e-services are federated in nature as they interact across management domains and enterprise networks. Their implementations could be vastly different in nature. They could be based on CORBA [4], BizTalk [3], COM, E-speak [2] or on other platforms. The diversity in their implementations makes it difficult to manage them. They also need to agree upon document exchange protocols to communicate and interoperate with each other. These E-services undertake *conversations* [5, 8] in particular agreed upon protocols like CBL, cXML, EDI etc. Every conversation consists of multiple exchange of XML documents, which are termed *interactions*, that involves exchange of documents. A conversation in the simplest case consists of a *transaction*. A conversation can in effect contain multiple or no transactions.

Management of E-services is a challenging task because of the varied, federated, decentralized and distributed nature of e-services. As a single transaction spans multiple e-services it is difficult to undertake end to end e-service management in a distributed and decentralized manner.

Motivations for end to end e-service management arise from the perspective of both clients and service providers. Clients are interested in tracking their interactions and in understanding the e-service process flow internals. This enables the client to seek some insight to the actual e-service flow. 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 e-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.

B. INTERACTIONS BETWEEN E-SERVICES

A typical e-service would get an http request from the parent e-service (if it itself is not the root service) or an ultimate consumer. The request is routed from one of the webserver in the webserver farm to one of the application modules in the application server farm. Thereafter, business logic is applied to the request and a new request can be directly sent to sub e-services

[1] HP Laboratories Palo-Alto

[2] HP OpenView Business Unit

through an http request at this level. The response from the sub e-service is expected at the listener (attached to the web server farm) that is waiting to receive documents.

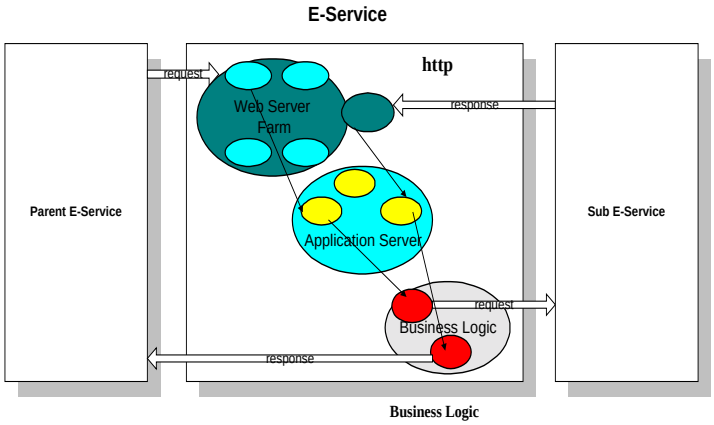

Figure 2. E-Service interactions

Once the corresponding sub e-service responds with a document the response is sent to the business logic, which in turn sends a response to the parent e-service. The communication pattern can vary depending on the implementation. For example, the application logic can send an immediate response to the parent e-service before receiving response from the sub e-service. The final response may be sent later. Also, the final response can be sent directly to the parent e-service by the sub e-service, instead of being routed through the business logic. In addition the format of communication is not symmetric in the e-services world. An E-service request may not always have a matching response. A single e-service request can have multiple responses.

As there is no single point of control it is difficult to monitor the request and responses going into and coming out of an e-service and correlate them without knowing the business logic of the E-service. E-service conversation and transaction level correlation would therefore need intrusive instrumentation.

Application Response Measurement (ARM) [7] is a standard in intrusive instrumentation of applications. Although ARM cannot be directly applied to e-service management domain it is interesting to understand the functioning of ARM. This will enable an understanding of the proposed distributed correlation approach.

C. ARM Overview

ARM provides APIs that can be used to delimit sections of application code base to monitor time spent in those sections. These APIs correspond to starting and stopping of code sections and assigning handles to them for manipulation by the local ARM Agent.

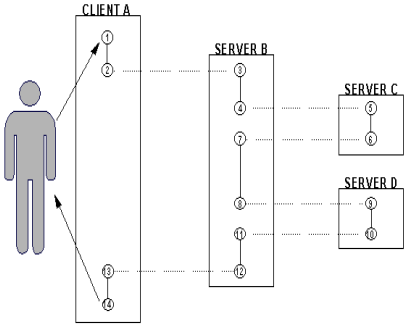

Figure 3. Usage of ARM

ARM 2.0 API also adds interesting functionality in the form of correlation and furnishing of application data in the form of data buffers that are maintained by the ARM library. The correlation data is sent to a central correlation application that undertakes the task of linking up the transactions with their component transactions.

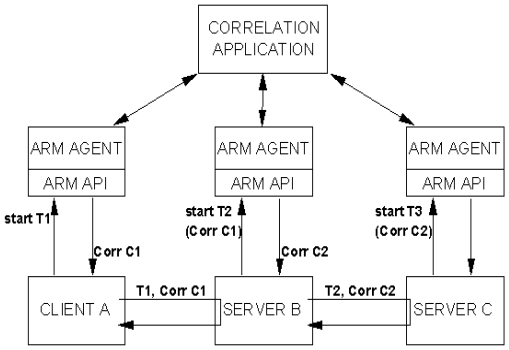

Figure 3. Correlation in ARM

An E-service on the other hand undertakes conversations with other e-services, which involves multiple interactions (exchange of documents). These e-services are federated and distributed with varied implementations. In addition, their interactions are asymmetric and asynchronous in nature. On the other hand ARM assumes a centralized correlation application. A concern with ARM when tracing any application flows is overwhelming the correlation application and/or the network with the volume of data collected at the agent, sent to the correlation application, and processed. ARM is also ill suited to the asynchronous and asymmetric interaction model that e-

services operate in. ARM is also designed to maintain transaction level data while e-services need conversation monitoring.

D. E-SERVICE MANAGEMENT THROUGH DISTRIBUTED CORRELATION

At any given instant of time e-services form a tree or a directed acyclic graph with e-services at various nodes and conversation related interactions (document exchanges) as the branches (refer figure 1).

As these DAGs are created either statically or dynamically and conversations/transactions are performed management information can be exchanged amongst these e-services. We term our protocol as the MI protocol (Management Information Protocol) a.k.a. *my protocol*. This Management Information (MI) would be exchanged amongst these e-services. The MI structure contains two types of data at every level: the correlator and the management information object (MIO) a.k.a. *my Objects*. As an e-service conversation can go across multiple e-services, the correlation information (correlator) is needed to track a conversation and correlate the management data collected at each participating e-service. As the conversations are performed the MI structures would be correlated and sent back along with the response documents. MIs at every level are all collated into a single MI structure finally in the response to the parent e-service. In other words, when an e-service sends its parent a response at the middle or the end of its conversation, the piggybacked MI contains the sub MIs relative to the sub e-services involved in the conversation.

The biggest problem in providing such a correlation technique is the fact that the request and response documents are usually different documents. The MI library locally maintains the correlation information and updates this in the header of the documents provided to it. In order for this scheme to work it would need cooperation from the e-services to provide the MI library with its document (it receives and intends to send) and uses the documents (with modified header) returned by the MI library instead.

The MI structures are created by default by the MI library and finally the document received by the parent e-service would contain the full correlator containing the transaction flow information.

E-services at each level can further enrich the data collection by agreeing on a certain ESSMIO (E-service specific MIO) defined in a particular schema that would enable the collaborating e-services to furnish business logic data (e.g. the number of book buy requests received etc). The default MI structure contains the correlator and a certain set of predefined management information data as described in section G.

1st. MI APIs

An e-service conversation/transaction usually starts with a request from a consumer or a parent e-service, and ends with a response or a new request to another e-service. In between, one or more of the following can occur, depending on how a conversation is defined and implemented.

- The e-service replies with an immediate response to a request (which will be an http response). The actual business level reply is sent later asynchronously;
- The e-service can receive one or more subsequent requests;
- The e-service can send one or more requests to one or more of its sub/external e-services;
- The e-service can receive one or more responses from its sub/external e-services.

To manage an e-service conversation with the above communication pattern, two issues must be addressed. First, the communication between e-services is asymmetric. That is, one or more responses can correspond to one request, or vice versa. Furthermore, an e-service can send a response directly to the ultimate requester, instead of its parent. Second, the request/response documents to be sent/received by an e-service can be handled by different threads/components while each thread/component handles the documents relative to different conversations. To address the two issues, some form of correlation is needed, and we propose a mechanism and an API on top of it for generating, exchanging, and syndicating MI structures at cooperating e-services. As a result, an e-service, when receiving a request/response, can associate it with a specific conversation, and from management point of view find out which response corresponds with which request.

MI APIs enable getting and setting of Management Information Objects.

Two types of information are collected from e-services. One type is *descriptive* information, which contains the e-service and transaction definitions. The other is *management* information, which contains the data for each instance of a conversation. The management data is part of the MI definition, which will be described in the following section.

Descriptive information is provided in the class MITranRegistration. An e-service, when getting started, initiates an instance of this class and calls registration method, MI_register_transaction, to define the mapping from a universally unique transaction class ID to a name pair (service, transaction). There are two versions of MI_register_transaction, depending on whether the transaction ID is provided as a parameter, or generated by the MI library. The MI library maintains a list of registered transactions. When the method MI_register_transaction is called, a new entry will be added in the list.

```
public class MITranRegistration extends Object {
// Public Constructors
        public   MITranRegistration();
// Public Instance Methods
        public short   MI_register_transaction(
                          String  service_URI,
                          String  tran_name,
                          byte[]  tran_id;
                          int     flags);
        public byte[]  MI_register_transaction(
                          String  service_URI,
                          String  tran_name,
                          int     flags);
// misc …
}
```

Management information is provided in the class MIServiceTran. This class represents e-service transactions when they execute. An e-service creates as many as instances it needs. This would typically be at least as many as the number of transactions that can be executing simultaneously. An e-service would typically create a pool of MIServiceTran objects, take one from the pool to use when a transaction starts, and put it back in the pool after the transaction ends for later reuse. Internally, each entry in the list of registered transaction has a list of transaction instances. Each entry contains the MI structure of a specific transaction instance. When a new instance of a transaction class is started, a new entry will be allocated and added in the corresponding instance list. The contained MI structure will be updated at each stage of the transaction. The maximum length of each transaction instance list depends on the system configuration. The definition of the class is as follows.

```
public class MIServiceTran extends Object {
// Public Constructores
  public MIServiceTran();
// Public Instance Methods
  public Policy MI_getPolicy
                    (Document document
                     int doc_type);
  public long MI_start(byte[] tran_id,
                     long ptran_handle,
                     ESSMIO essmi_object,
                     int flags);
  public long MI_start(byte[] tran_id,
                     long ptran_handle,
                     long tran_handle,
                     ESSMIO essmi_object,
                     int flags);
public long MI_update(long tran_handle,

        ESSMIO essmi_object,
                     int flags);

  public Document MI_sendReq
                    (Document document,
                     int doc_type,
                     long tran_handle,
                     Policy req_policy,
                     ESSMIO essmi_object,
                     int flags);
  public long MI_recvReq
                    (Document document,
                     int doc_type,
                     long tran_handle,
                     ESSMIO essmi_object,
                     int flags);
  public Document MI_sendRep
                     (Document document,
                     int doc_type,
                     long tran_handle,
                     ESSMIO essmi_object,
                     int flags);
  public long MI_recvRep
                    (Document document,
                     int doc_type,
                     long tran_handle,
                     ESSMIO essmi_object,
                     int flags);
  public long MI_stop(long tran_handle,
                     int tran_status,
                     ESSMIO essmi_object,
```

int flags);

// misc for getting MIO and ESSMIO…
}

If a transaction is due to a request (i.e., a XML document) from a parent e-service or a consumer, MI_getPolicy is invoked to retrieve from the document the policy containing the information about how the service request should be handled from the client's perspective. Method MI_start is used to start a transaction. There are two versions of MI_start, depending on whether a transaction handle is provided by the user or generated by the MI library. The other parameters to this method are the transaction class ID, the parent transaction handle, and the ESSMIO. The transaction ID is the type of transaction this transaction instance (the transaction handle) belongs to. The parent transaction handle associates this transaction with its parent transaction if any. The ESSMIO is used to contain the necessary e-service specific management information for this transaction, if any.

If the transaction is due to an external service request, it calls method MI_recvReq by providing the transaction handle and the received document. MI_recvReq retrieves the MI and ESSMIO trees from the document header, and creates a new MI structure containing the correlator and management information for this transaction instance. Then MI_recvReq creates a new MI/ESSMIO tree by appending the transaction's MI/ESSMIO to the MI/ESSMIO tree retrieved from the request, and associates the paths with the transaction instance. The resulting MI and ESSMIO trees are used to identify this transaction instance in the context of the containing e-service conversation and present the corresponding management statistics. The correlator for a transaction instance is made unique by the combination of

- The transaction class ID
- The transaction handle, an incremented integer
- The interaction handle, an incremented integer, corresponding to an interaction with a sub e-service

The IDs and handles of transactions participating in a conversation are combined to identify the conversation. The interaction handles of a specific transaction instance are used to identify the sub e-services' contributions to the transaction's statistics.

As described before, an e-service can receive a subsequent request from its parent during a transaction. When this occurs, MI_recvReq is invoked by providing the same transaction handle, the new received document, and the updated ESSMIO of the transaction if any. Then MI_recvReq is executed to update the transaction's context (i.e., its MI and ESSMIO trees) with the retrieved and passed MIs and ESSMIOs.

Method MI_sendReq is called when an e-service needs to send a service request to a sub e-service in one of the following scenarios: after receiving a request from its parent (MI_start and/or MI_recvReq was called), or after receiving a response from a sub e-service (MI_recvRep was called). The parameters to this method are the document to be sent, the transaction handle, and the updated ESSMIO if the transaction's e-service specific management information has changed. A policy can optionally be provided to indicate how the request should be serviced. The transaction handle is used to locate the MI and ESSMIO trees of the transaction. If a policy and/or an updated ESSMIO is supplied, MI_sendReq will use them to update the transaction's MI structure and/or ESSMIO accordingly. MI_sendReq is invoked to perform two tasks: first, the MI library records the start of a new service interaction for this transaction instance; second, generates the proper MI and ESSMIO trees for the outgoing interaction, which should include the MIs/ESSMIOs for this transaction instance and all of its predecessors, and exclude those relative to the sub e-services the transaction previously interacted with. Then MI_sendReq inserts the generated MI and ESSMIO trees into the header the document to be sent, and returns the document to the caller.

When a transaction receives a response from an external e-service, it calls MI_recvRep to mark the end of the e-service interaction. MI_recvRep is executed to retrieve the MI and ESSMIO trees containing the latest conversation path and the management statistics relative to the latest interaction. Then MI_recvRep uses the transaction handle to locate the transaction's entry, marks the end of the interaction, and uses the retrieved MIs and ESSMIOs and the passed ESSMIO to update the transaction's statistics.

An e-service may send back some preliminary or final result before or at the end of the transaction. If this occurs, the current conversation path and its statistics at each participating e-service tier need to be sent back together with the response document. Also, the transaction's context needs to be updated so that the subsequent response will not contain the statistics and e-service interactions prior to this point. To achieve this, method MI_sendRep is invoked by providing the document to be sent, the transaction handle, and the transaction's ESSMIO if updated. With the transaction

handle, MI_sendRep locates the transaction's context, and updates and summarizes its statistics. Then it generates the MI and ESSMIO trees containing the transaction's statistics and the breakdowns at each participating e-service tier. Finally MI_sendRep inserts the generated MI and ESSMIO trees into the header of the response document before it is sent back.

MI_end is used to inform the MI library the end of a transaction, and summarizes its statistics. It also informs the MI library the status of this completed transaction. If the application logic finishes successfully, the status is set to MI_GOOD. If the service request is not satisfied (e.g., could not reserve a hotel room due to no vacancy), the status is set to MI_ABORTED. The status is set to MI_FAILED if there is an application or system failure (e.g., a sub service is unavailable).

2nd. Usage of MI APIs

The APIs provided by the MI library can be classified into three categories: registration, demarcating business transactions, and tracking interactions/conversations with the outside world. When an application/e-service starts, it calls MI_register_transaction to register the types of transactions it provides.

During runtime, it calls MI_start to begin an instance of a registered transaction, and calls MI_stop to end the transaction instance. In between the application/e-service, calls MI_sendReq, MI_recvReq, MI_sendRep, or MI_recvRep when sending or receiving a request or response from the outside world.

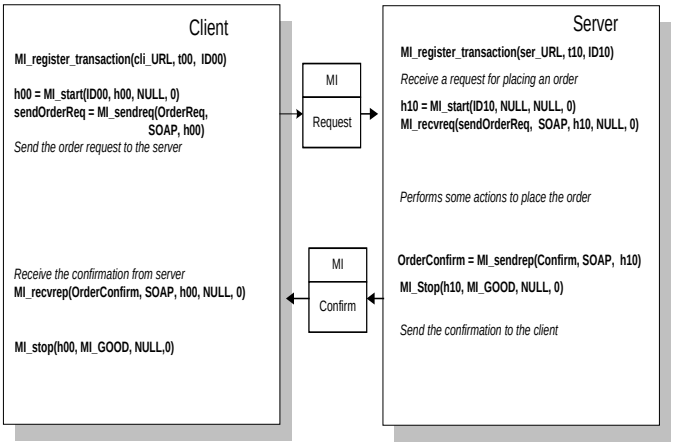

Figure 4. A service request for placing an order

Figure 4 shows an example where the client sends a service request for placing an order, and the server replies with a confirmation after processing the request.

A more realistic example would a travel e-service that interacts with air flight, hotel, and car rental e-services. The interactions between these e-services are shown in figure 5. The travel e-services undertakes conversations and exchanges documents with Airline e-service, Hotel E-service and Car e-service.

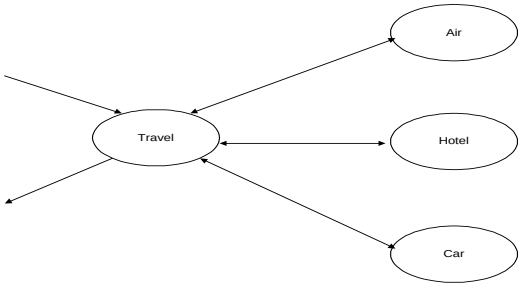

Figure 5. The interactions between a travel e-service, its consumer, and its sub-e-services

Figure 6 illustrates how the MI API is used to instrument the travel and its sub e-services. Note that, though the conversation in the example is symmetric, the MI library and its API also support asymmetric conversation model. Moreover, a transaction can go across different threads/components. In other words, MI_start, MI_recvReq, MI_sendReq, MI_sendRep, MI_recvRep, and MI_stop for one transaction can be called by different threads/components.

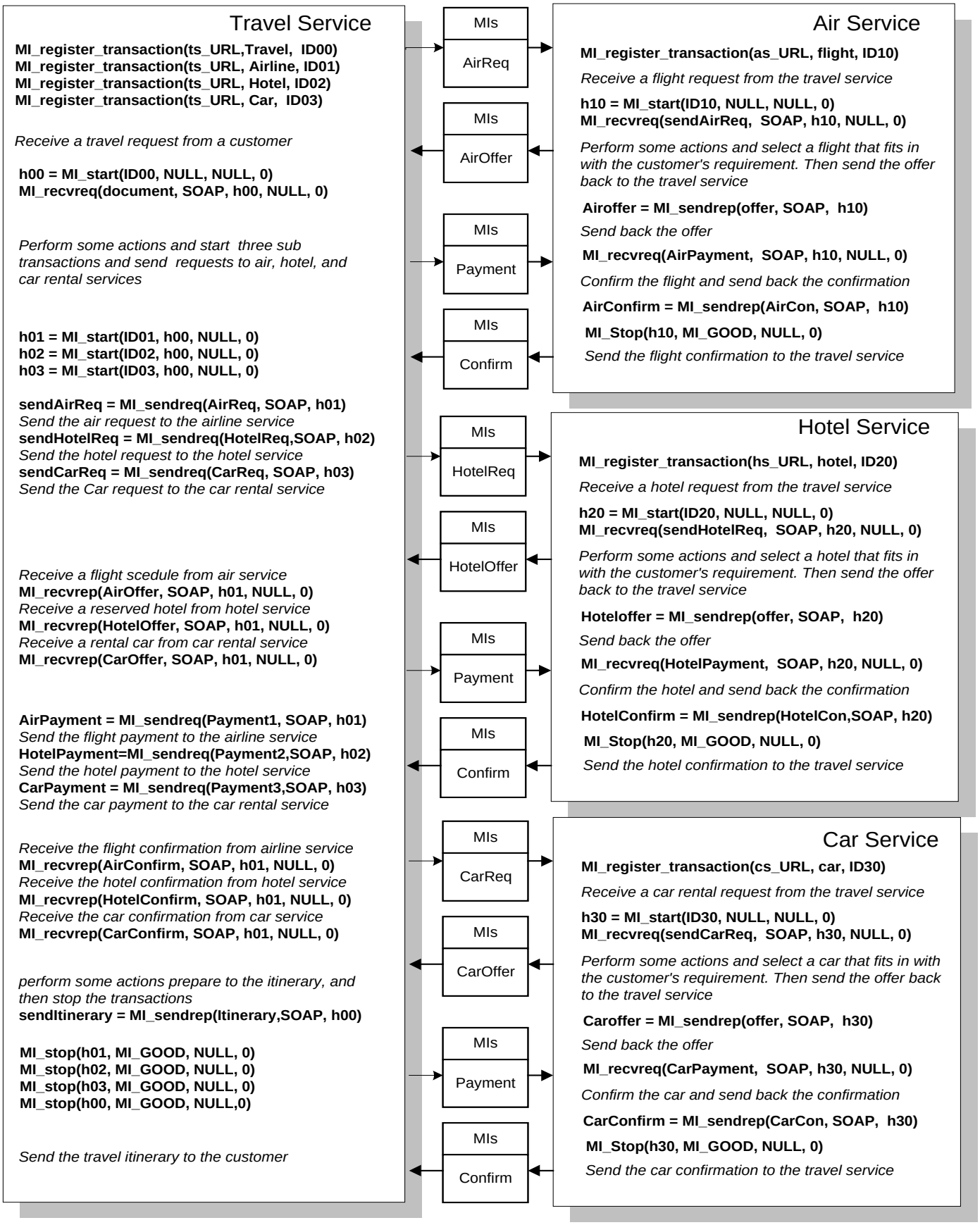

Figure 6. An instrumented travel e-service and its sub e-services

E. MI DATA FORMAT DEFINITION

An MI *structure* contains a correlator and a management information object (MIO), and the MIO may consist of a policy and a set of management parameters. Figure 7 shows the MI structure. We have described the format of the correlator and its use for measuring end-to-end e-service conversation and the breakdowns at each participating e-service tier. In this section, we derive a set of management parameter [6] that should be included in the MIO, then present the MI schema [1].

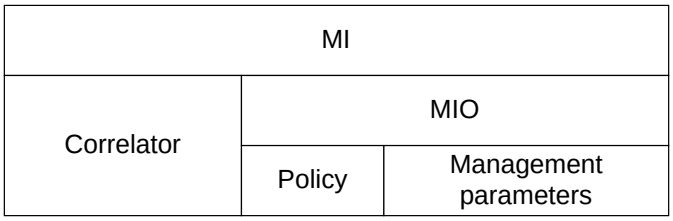

Figure 7. The MI structure

The management parameters in MIOs are classified into the following types. Note that the first type of parameter is provided by the client and sent to the server, and the other types of parameters are provided by the server and sent to the client.

- SLA related policies. This type of parameter presents the expectation from a consumer or an e-service how the service request should be handled. For instance, a request can be prioritized as "normal", "silver", "gold", or "platinum", and is expected to be serviced within a certain amount of time under certain constraints (SLO).
- E-service health index. This indicates the current status of an e-service. Possible values are: *up, down, congested, halted, restarting,* and *unknown.*
- Availability. This indicates an e-service's availability, in terms of downtime per a certain period of time. Downtime is the duration of e-service unavailability due to system or application faults.
- Reliability. The reliability of an e-service can be measured in terms of the fault rate. That is, the number of faults per the number of handled service requests.
- Performance. Performance parameters include response time, response status, throughput/rate, aborted count/rate, and failed count/rate. Response time is the duration from the sending of a request to the receiving of a response. Response status indicates if a service request has been done, aborted, or failed. Throughput/aborted/failed rate is the number of committed/aborted/failed services per the total number of service requests.
- Faults. A fault parameter contains the information about a fault when a request is serviced (e.g., fault ID, fault description, time of the fault occurred).

The above parameters form a base for the management of e-service conversations. Depending on how much information one e-service is willing to expose to another, some of the raw parameters can be put into MIOs—the aggregated information can be calculated based on the raw parameters, and need not be put into MIOs. For instance, the MI library can get a throughput rate of a sub e-service by using the formula: the number of successful responses divided by the number of total responses from the sub e-service.

By aggregating and analyzing MIOs, the MI library can provide the necessary information that higher level management agents can use to perform the following management tasks. Firstly it can be used for end-to-end conversation management. It enables an e-service or a consumer to track its interactions with other e-services and the e-service flow. Secondly, it helps an e-service perform e-e-service ranking and selection. An e-service can rank its external e-service providers based on their performance, availability, and reliability. The latest statistics of the external e-services can also help the e-service choose the right e-service supplier. Thirdly it enables service optimization. By identifying performance bottlenecks and service failure points, the management agent can reconfigure and/or restart the e-service to achieve better performance and availability. It also helps in e-service evolution. An e-service provider usually provides different types of services. The information provided by the MI library could be used to monitor the access patterns for the provided services, and help make decisions about whether or how to evolve the non-performing services to generate more revenue. Lastly it enables consumer tracking. The information can be used to find out the consumer pattern so that actions can be taken to guarantee the QOS for valued consumers.

3rd. The MI schema

Each MI structure contains a correlator identifying the context of a transaction in a conversation. It is transparently handled by the MI library. Its schema is:

```
<complexType name = "CorrelatorType">
  <element   name = "TranID" type = "string"/>
  <element   name = "TranHandle"
             type = "decimal"/>
```

```
  <element   name = "InteractionID"
             type = "decimal"/>
</complexType>
```

A policy sets a priority for a service request. It can also set a set of QoS paramenters (SLOs) indicating how and when the request should be serviced.

```
<complexType name = "PolicyType">
  <element name = "Priority" type  = "decimal"
      minOccurs = "0" maxOccurs = "1">
  <element name = "SLO"      type = "SLOType"
      minOccurs = "0" maxOccurs = "unbounded">
</complexType>
```

```
<complexType name = "SLOType">
  <element name = "Term"  type = "string"/>
  <element name = "Constraint" type="string"/>
  <element name = "Threshold"  type="string"/>
</complexType>
```

Currently the measurement data in each MIO contain the identity (i.e., URI for a consumer or service provider), and performance, availability, and reliability statistics.

```
<complexType name = "MeasurementType">
  <element name = "Identity" type = "IDType"/>
  <element name = "Performance"
           type = "PerfType"/>
  <element name = "Availability"
           type = "AvailType"/>
</complexType>
```

```
<complexType name = "IDType">
  <element name = "From" type = "serviceURI"/>
  <element name = "to"   type = "serviceURI"/>
</complexType>

<complexType name = "PerfType">
  <element name = "RespTime" type="decimal"/>
  <element name = "StartTime" type="decimal"/>
  <element name = "StopTime"  type="decimal"/>
  <element name = "TranStatus"
           type = "decimal"/>
</complexType>

<complexType name = "AvailType">
  <element name = "ReqCount" type ="decimal"/>
  <element name = "ComCount" type ="decimal"/>
  <element name = "FailCount" type="decimal"/>
  <element name = "AbortCount"
           type = "decimal"/>
</complexType>
```

The MIO consists of Policy and MeasuredData. Its schema is as follows.

```
<complexType name = "MIOType" >
  <element name = "Policy" type="PolicyType"
      minOccurs = "0" maxOccurs = "1">
  <element name = "MeasuredData"
           type = "MeasurementType"
      minOccurs = "0" maxOccurs = "1">
</complexType>
```

The Correlator and MIO construct a MI structure that is used to identify a transaction's local context and statistics.

```
<complexType name = "MIType" >
  <element name = "ParentCorrelator"
           type = "CorrelatorType"
           minOccurs = "0" maxOccurs = "1">
  <element name = "Correlator"
           type = "CorrelatorType"/>
  <element name = "MIO" type="MIOType"/>
</complexType>
```

To identify a service request in the context of a converstaion, an e-service needs to get the MI structures of its predecessors. To know how its sub e-services perform, an e-service needs to get the MIs of the participating e-services. MITree is defined to serve the need. This is the structure piggybacked on the documents exchanged between e-services.

```
<element name = "MITree"
         type ="MITreeType"/>

<complexType name = "MITreeType" >
  <element name = "predecessor"
           type = "MIType"
      minOccurs = "0" maxOccurs = "unbounded">
  <element name = "MI"
           type = "MIType"/>
  <element name = "Child"
           type = "MIType"
      minOccurs = "0" maxOccurs = "unbounded">
</complexType>
```

F. Evaluation of the Distributed Correlation Approach

This approach has multiple advantages and certain disadvantages

4th. Advantages

- Information containment at every level

e-services have control over the information they reveal to other e-services. Depending on the negotiated manageability they reveal information to other e-services.

- Uniformizing different e-service implementations

Irrespective of different and varied implementation of e-services, if they agreed on certain data formats for MI structures to be exchanged these e-services could be managed

- Policy specified by requester

The requesting e-service can specify a management policy for handling of a conversation document in the MI structure. This could be ignored or taken into account by the receiving e-service and will enable it to handle them differently according to the policy specified in the MI structure of the received document. The E-

service specific MIOs can be used to quantify these service level expectations

- Saving on http connection by piggybacking

MIs and ESSMIOs are sent back piggybacking on the responses and no special connections are made to a management system at every e-service level. There is no central correlation agent who is entrusted with the job of maintaining the correlations. The data collection is decentralized.

- Support for multiple conversation formats

This protocol is capable of handling symmetric, asymmetric and synchronous, asynchronous conversations.

5th. Disadvantages

There are also certain disadvantages to this approach

- The MI structures are sent back with the response documents that can increase their length depending on the level of nesting allowed.
- E-services do not have a global picture till the response is sent back. In case the response is not sent back immediately the management information is not conveyed back to the previous e-service till the whole chain of transactions is complete.

G. ACKNOWLEDGEMENT

We would like to thank Sekhar Sarukkai for having contributed to the initial concept of distributed correlation.

H. CONCLUSION

End to end E-service transaction and conversation management is a challenging task. The distributed correlation approach enables this task in a distributed and decentralized manner. The data collected by this approach can be used for business logic optimization and e-service management.

I. REFERENCES

[1] XML at World Wide Web (WWW) Consortium. http://www.w3.org/xml

[2] Hewlett-Packard Company. E-Speak Architecture Specification. Version Beta2.2. December 1999. http://www.e-speak.net/library/pdfs/E-speakArch.pdf

[3] D. Rogers. *BizTalk service framework*. Microsoft Corporation. http://www.biztalk.org

[4] Object Management Group. *The common object request broker: Architecture and specification*. Revision 2.0, July 1995 http://www.omg.org

[5] A. Dan and F. Parr. *An Object implementation of network centric business service application (NCBSAs): conversational service transactions, service monitor, and an application style*. OOPSLA'97, Business Object Workshop III.

[6] J. T. Park and J. W. Baek. *Web-based Internet/Intranet service management with QoS support*. IEICE Trans. Commun., e82-b:11, 1999.

[7] ARM Working Group. Application Response Measurement API Guide. 1997. http://www.omg.org/regions/cmgarmw/index.html

[8] K. Evans, J. Klein, and J. Lyon. *Transaction Internet Protocol – Requirements and Supplemental Information*. 1998. http://www.landfield.com/rfcs/rfc2372.html