SOA Without Web Services: a Pragmatic Implementation of SOA for Financial Transactions Systems

Ziyang Duan, Subhra Bose
Reuters America
3 Times Square, 18th Floor
New York, NY 10036
{ziyang.duan, subhra.bose}@reuters.com

Paul A. Stirpe
New York Institute of Technology
Computer Science
Old Westbury, New York 11568
pstirpe@nyit.edu

Charles Shoniregun
University of East London
School of Computing & Technology
Essex, RM8 2AS UK
C.Shoniregun@uel.ac.uk

Alex Logvynovskiy
Business, Computing & Information Management
London South Bank University
103 Borough Road
SE1 0AA, UK

Abstract

The Service Oriented Architecture (SOA) provides a methodology for designing software systems by integrating loosely coupled services. Compared to traditional distributed object-oriented architectures, SOA is more suitable to integrate heterogeneous systems, and more adaptable in a changing environment. This paper presents the design and implementation of a SOA framework for financial transaction applications. The framework provides an easy and uniform way for service composition in a controlled environment, and leverages Web service standards with efficient communication mechanisms and durable and/or transactional message queues. Specifically, the work addresses the following issues: 1) the incorporation of existing systems and protocols that are not Web-service compatible. This paper focuses on business processes of equities transactions using the FIX *[6] protocol. 2) the configuration and deployment of services and service endpoints in a flexible and dynamic manner. 3) the capability of specifying business processes as Web service compositions and a distributed runtime environment that supports it. 4) the scalability, resiliency and transactional aspects as required in critical business applications. The experience of applying the framework in building a high performance equities transaction system is presented.*

1 Introduction

The financial industry demands efficient, affordable, and flexible approaches to automate and integrate their business processes. To meet this challenge, Service Oriented Architecture (SOA) is becoming a favorite choice of software architects who struggle to provide solutions for distributed applications, while maintaining manageable system architectures. SOA's building block is a set of loosely-coupled services. A service provides a unit of functionality by exposing its abstract interface on the network. The business functionality is implemented as coordinated interactions of services. SOA allows heterogeneous components to be easily integrated to satisfy business requirements. Furthermore, such systems are more flexible and adaptable than traditional object-oriented distributed systems built upon strongly coupled components [12].

Many XML-based standards have been widely adopted to support Web service specifications, interactions, discovery and composition. For example, WSDL[3] provides for interface definition and binding, and uses UDDI for service discovery, SOAP [7] for message exchange, and BPEL[4] for service composition. These standards lay out the foundation of Web service technologies. Major development tools and middlewares now offer supports to those standards. With the help of these tools, users can now conveniently create and deploy Web services, or expose the interface of an existing application as a Web service.

Recently, Web services and service-oriented architecture have been applied to different areas. For example, Zimmermann *et al.* [14] developed a Web services-based system to integrate inter-organizational banking systems. Petinot [11] proposed a service-oriented architecture for building semantic-based information retrieval systems. Zhang *et al.* [13] developed a pass-through authentication Web service framework for on-line electronic payment applications. Medjahed *et al.* [8] implemented an ontology-based web service composition framework for e-government system. Fileto *et al.* [5] described a semantic-based approach for Web service composition and its application in agriculture planning.

Many commercial products have been developed to automate business process integration process through Web service integration, for example, IBM's Websphere, BEA's Web logic, Microsofts Biztalk, etc. These products provide intuitive process design tools, deployment infrastructure, and execution environments integrated with available application servers and workflow systems, It is easy to use these tools to design and deploy a business process by integrating existing Web services together. Therefore they are increasingly adopted in many application domains, especially those that demand a timely solution.

However, there are still many issues remaining to be resolved in building high performance financial transaction systems using the existing technologies [2, 1]. Current technologies focus on providing good interoperability over the Internet using standard protocols such as WSDL and SOAP, whereas existing financial systems are usually based on proprietary protocols and message formats. A Web service-based system would have to translate messages from their native format to SOAP and vice versa in order to facilitate interoperability. Furthermore, the protocols usually define their own way to handle various aspects of interactions, such as authentication, reliability, maintaining sessions and states, failure-handling, etc. To incorporate them into the existing Web service framework would be either impossible or produce inefficient solutions. Protocols, such as *FIX* [6], are widely adopted and serve as the *de facto* industry standards. Thus, it is impossible to replace them with newly-designed, Web service-compatible protocols in the foreseeable future. In addition, the protocols and message formats are usually designed for efficient transmission and parsing, which is important in systems requiring high throughput and high availability. It is difficult to satisfy these requirements by using SOAP and WSDL-based services, because processing XML-based messages introduces additional latency and bandwidth cost.

Reliability and transaction support are major concerns in financial transactions systems. Though many efforts are focused on addressing this issue, such as the proposal of WS-Security [9] and WS-Transaction [10] standards, the support of such features in existing web service integration frameworks are still very primitive.

Large-scale financial transaction systems require high throughput and scalability. At periods of peak load, the systems are required to handle thousands of concurrent requests with latencies typically less than a second. Program trading applications demand even higher performance requirements. However, current Web service integration frameworks introduce additional layers and components such as web server, SOAP translation, WSDL mapping, that require more computation and introduce additional latency. Though most Web service integration tools provides scalability solutions, they usually require more computing power to achieve the same level of performance compared to a traditional solution. This ultimately means higher cost in terms of deployment and maintenance.

We designed a SOA framework for building financial transactions systems with the above issues in mind. The framework provides an easy and uniform way for service composition in a controlled environment. In our framework, we assume services are deployed within an enterprise environment and the message payloads follow a standard, such as FIX (Financial Information Exchange). The framework leverages web service standards on top of efficient communication mechanisms, such as durable and/or transactional message queues. Thus, messages can be transported in a more efficient manner. The framework allows existing legacy systems based on the FIX protocol to be incorporated in our system, as well as Web service-based systems. The framework provides scalability, resiliency and transactional aspects as required in critical business applications. We also present our experience in applying the framework in building a high performance financial transactions system.

2 FIX protocol and equity transactions

The FIX (*F*inancial *I*nformation e*X*change) protocol is a standard to facilitate electronic communications of trade related messages. The protocol is developed as a collaboration effort among major financial institutions, exchanges, brokers, and information technology providers. It has been widely adopted in automated trading systems. FIX originated as a protocol to support equity transactions in the US market, and has been evolved to support international trading and more trading types.

A FIX message is a sequence of $< tag >=< value >$ fields delimited by the ASCII SOH (0x001) character. The fields can be partitioned into three parts: a header, a body, and a trailer. The tags are integers that represent predefined field names. Each message has a message type field (tag 35) in the standard header. For example, `35=A` means a logon message, `35=7` means advertisement, `35=6` means indication of interest, `35=D` stands for a single new order. Each type of message has a set of predefined mandatory and optional fields in the message body, and users can also define their own customized message fields.

Example 2.1 (Example of a FIX message)

```
8=FIX.4.0#9=0168#35=6#49=A#56=B#34=85#52=20050110-13:00:00
#23=13#28=N#54=1#27=1000#62=20040110-14:00:00#44=25.00
#15=USD#55=MSFT#10=130
```

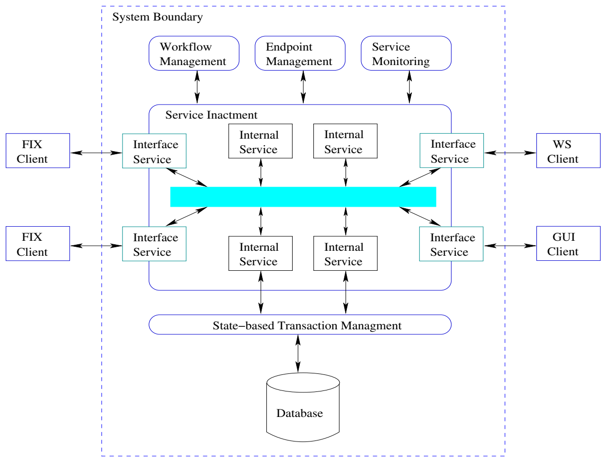
◇

Example 2.1 is an Indication Of Interest (IOI) message sent from A to B that indicates an interest of buying 1000 shares of Mocrosoft stocks at price of $25.00. The interpretation of each field is shown in table 1.

Tag	Field	value
8	version	"FIX.4.0"
9	body length	168
35	message type	IOI
49	sender comp ID	A
56	target comp D	B
34	sequence number	85
52	sending time	2005/1/10 13:00:00
23	IOI ID	13
28	transaction type	N (new)
54	side	buy
27	shares	1000
62	valid until	2004/1/10 14:00:00
44	price	25:00
15	currency	US Dollar
55	symbol	MSFT
10	checksum	130

A FIX session maintains a dedicated and reliable connection between two FIX applications, or FIX endpoints. All FIX messages are identified by a unique sequence number during a session. Sequence numbers are initialized when a FIX session is established, and increment through the session. The number is used to identify missed messages, and synchronize applications during reconnection. Thus, messages delivery can be considered completely ordered. FIX applications use *heartbeat* messages at regular intervals to monitor the status of communication link. In addition, the FIX protocol provides mechanisms for authentication, encryption, and message recovery.

The framework provides a platform for FIX-based equity transactions. Our platform connects hundreds of major financial institutions and brokers located globally through thousands of dedicated FIX endpoints, and processes tens of millions of FIX messages per day. Business processes of equity transactions are implemented as sequences of FIX messages flowing between the trading partners. The processes are composed of pre-ordering activities such as indication of interest and advertised traders, order processing activities such as create new order, order canceling and replacement, and post ordering activities such as execution



reporting. The platform serves as a hub for equity trading processes. Specifically, the platform provides the following functionalities:

- Maintain dedicated FIX connections.
- Dynamically add new and configure existing endpoints.
- Route messages to corresponding services and process messages according to the business logics.
- Add new services and functionalities and customize existing services according to the business requirement.
- Provide reliability and transactional correctness guarantee for business processes.
- Provide failure recovery, auditing, and monitoring capabilities.

3 The SOA Framework

Architecture Overview

Figure 1 shows the overview of our system architecture. The system is composed of the following components.

- *Service Enactment* provides the core functionalities for service invocation and message routing.
- *Workflow Management* provides tools to specify flow logics as service composition, and to compile the workflows into a set of message routing rules that can be used by the service enactment component.
- *State-based Transaction Management* guarantees transactional correctness of business objects based on their state chart model.
- *Service monitoring* provides tools to view the execution status of services, and handling failure situations.

- *Service Configuration* provides tools to dynamically add and reconfigure service endpoints.

We hereby discuss the service management, workflow management, and transaction management in detail.

Services The service enactment component hosts a set of *services*. As shown in figure 2, a service is defined by its interface definition, the physical implementation, and the binding between them. The interface of a service specifies how a consumer should interact with it. It is platform-neutral and independent of its implementation in order to achieve maximum interoperability. The interface is specified following the WSDL standard. An interface definition specifies the set of *operations* provided by the service. Each operation is defined by the sequence of messages it sends and receives.

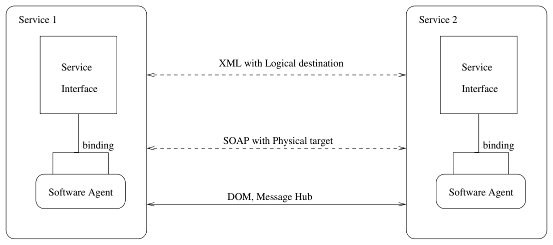
A service is physically implemented as a *software agent*. An interface is bound to a concrete implementation and message format via *bindings*. The interface of a service is then accessible via one or more *endpoints* based on the binding.

In order to achieve high efficiency for service communications within the system boundary, and provide maximum interoperability with existing FIX-based trading systems, services are distinguished into two types: *interface services* and *internal services*, as shown in figure 1.

An *internal service* implements a functional unit of business logic. Internal services only interact with other internal or interface services, and do not communicate with clients outside of the system boundary. All internal services consume and produce messages in a predefined internal message format. Internal messages are transported reliably through the *message hub*, which is discussed later.

The *interface services* are responsible for communications with other systems and clients. The interface services expose the functionality of the system in different ways. The system currently provide three types of interface services: FIX endpoints that implement the FIX protocol and communicate with FIX-based clients, Web service-based interface services that allow Web service-based clients to communicate with our system, and GUI-based interface services that allow users to interact with the system manually via a Web browser.

The interface services translate incoming requests into internal message format, and dispatch the requests to the corresponding internal services. In addition, they translate response messages to the formats compatible with the corresponding client APIs and legacy systems and send them to the target.



Messages A message definition defines the abstract types of messages exchanged between the service provider and the consumer. In our system, a message is a SOAP document that follows a predefined schema. As shown in figure 3, the SOAP header contains the following information:

- The type of the business process to which the messages belongs,
- The service name, operation name, and port type of the source and target service,
- Other optional information relevant to routing.

The SOAP body contains a sequence of transaction messages. Each message has a timestamp and a unique transaction ID for transactional control purposes. Specifically, FIX messages are represented in FixML.

To separate the logical design and physical implementation of a SOA based system, the message-based communication mechanism is abstracted into layers: the logical layer, the physical layer, and the binding between them. As illustrated in figure2, the abstract definition of a message is defined by its type in XML Schema. A physical instance of a message is either a native DOM object or an XML document compatible to the abstract definition. The source and destination of a message are the logical address of the service interface, which can be mapped into physical service instances.

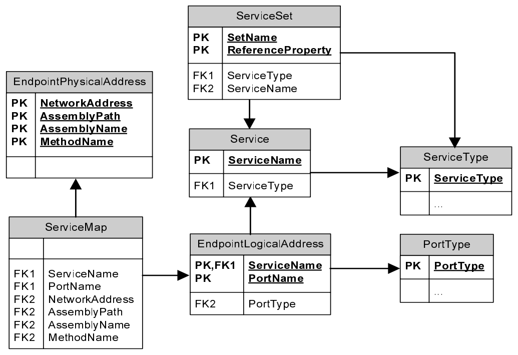
Endpoint addressing Each service has a logical address and a physical address. Logically, a service is identified by its name and characterized by its service type. The service type specifies the set of operations that the service supports. Each operation has one or more ports, which corresponds to the end points of message communication. A port is identified by a unique name within the service, and is associated with a port type that specifies the allowed type of messages and communication patterns. Each service in the system has one or more endpoints that sends and receives messages. Thus, an endpoint is uniquely defined by the corresponding

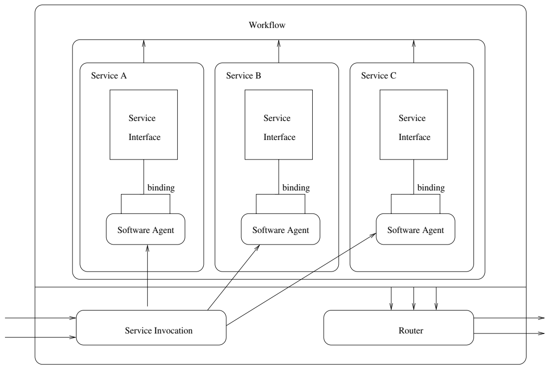
```
<SOAP-ENV:Envelope>
  <SOAP-ENV:Header>
     <Type>CreateIOI</Type>
     <TargetService name="IOI" operation="NewIOI"
                portType="SendViaFIX"/>
     <SourceService name="IOI" operation="NewIOI"
                portType="SendViaFIX"/>
     <ReferenceProperties>A,B,C, ... </
               ReferenceProperties>
     <!-- other application-specific routing-
               relevant properties -->
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <SOAP-ENC:string>
      <Transact>
        <Header ReqID="12345" TimpStamp="
               2005-01-10␣09:30:00" TransactID="
               6789" />
        <Body type=IOI>
          <fixml:IOI IOIID="100" TransType="N"
               Qty="99000" Side="1" Symbol="IBM">
             ... ...
          </fixml:IOI>
        </Body>
      </Transact>
      <Transact> ... </Transact>
    </SOAP-ENC:string>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

service name, service type, port name, and port type. Optionally, services can be grouped into service sets. Services in the same set have the same service type and therefore support the same set of operations. Each service in the set can be uniquely identified by the set name and the reference property field. A message can be multicast to several end points of services in the same set.

The physical address of a service identifies the location of the software agent that implements it. The endpoints are bound to the operations supported by the agent. In case of a Web service, the endpoints are identified by their URL. However, in our case, the software agents generally do not communicate through web service interfaces. It is preferred to interact via their native interface to improve performance. For example, many internal services in the system are implemented as .Net libraries. The physical address of such services are defined by the network address, the file path, the assembly name, and the operation name.

A logical endpoint is mapped into one or more physical endpoints in implementation. A logically unique service may correspond to several physically deployed software agents. Thus messages can be load-balanced to improve system performance. The mapping from logical service addresses to physical addresses is specified in the *ServiceMapping table*. The relationships are illustrated in figure 4.





Message hub Messages are delivered asynchronously to their destinations through the message hub. The message hub is implemented using transactional message queues. The message queues are communication channels for sending and receiving messages. Each physical service is mapped to a message queue from where it receives messages. A message delivered to a service is put into the correspondent message queue. Each queue has a unique network address and name. Services deployed at the same server location can share a message queue, but services at different server locations must use different queues. Since queues are not shared by locations, the address of a queue is also viewed as the address of the services that map to the queue.

As shown is Figure 5, at each physical location, the service enactment framework provides a *router* component and a *service invocation (SI)* component to handle the sending and receiving of messages, and dispatch them to the target endpoints through the message hub.

Service invocation The SI component is responsible for picking up incoming messages from a queue. Based on the logical address information in the header, SI finds out the physical address of the target endpoint, *i.e.*, the assembly and method name to be invoked, and then invokes the method to process the message (see algorithm 3.1) q .

Algorithm 3.1 *Service invocation*

```
ServiceInvocation(Message msg)
  read ServiceName, Operation, PortType from msg;
  PhysicalService ps := ServiceMap.find(
               ServiceName, Operation, PortType);
  if (ps does not exist) error_handling;
  if (NOT ps.Started)
    ps.Start();
    ps.invoke(ps.methodName, msg);
return;
```

Message routing The router is responsible for sending outgoing messages to their corresponding queues of the targeting services. The routing logic is stored in the *routing table*. Each entry in the table specifies a routing *rule*. Rules in the routing table are 6-tuples $< Type, Source, Pred, Target, Update >$, where

- $Type$ is the type of the business process,
- $Source$ is the logical address of the service that generates this message,
- $Pred$ is a predicate that evaluates to either true or false on the message header,
- $Target$ is the logical address of the target service, and
- $Update$ specifies an update operation on the message header.

Algorithm 3.2 shows the routing logic. For each outgoing message, the router looks up all the entries in the routing table whose $type$ entry match the "Type" field in the message header, and whose $source$ field match the target address in the header. The $Pred$ of each matched entry is then evaluated on the header. If the result is true, the router will update the target address in the header to $Target$, the source address to $Source$, and execute the $Update$ on the header. The message is then sent to $Target$.

To send the message, the physical address is first found in the *ServiceMapping* table. If more than one physical service is available, the router selects one of them based on a load balancing algorithm. In some cases, it is necessary to multicast a message to several different destinations. In this case, the $Target$ field contains the name of the *ServiceSet* that contains the logical addresses of all the destinations.

Algorithm 3.2 *Message routing*

```
Routing(Message msg)
  Ruleset =  ∀ rule ∈ routing table {\it s.t.}
             rule.Type = Msg.Header.Type and
             rule.Source = Msg.Header.Target;
  foreach rule in Ruleset
    if (Evaluate(rule.Pred, Msg.Header) = true)
         OutMsg = Msg.Clone();
      OutMsg.Header.Source = Msg.Header.Target;
      OutMsg.Header.Target = rule.Target;
      Evaluate(rule.Update, OutMsg.Header);
      SendMessage(OutMsg)
    end if
  end foreach
return
```

```
<process name="CreateIOI">
  <message name="IOI" type="NewIOI"/>
  <properties>
     <property name="sender" message="IOI" xpath=
              "/fixml/@senderCompId"/>
     <property name="target" message="IOI" xpath=
              "fixml/@targetCompId"/>
  <properties>
  <sequence>
    <receive msg="IOI"/>
    <foreach IOIMsg in IOI>
      <sequence>
         <invoke>IOICreate</invoke>
            <foreach name="destination" from="
               IOIMsg/target">
               <invoke  name="IOISend" message="
               IOIMsg" target="destination"/>
            </foreach>
         <invoke name="SendStreaming" message="
               IOIMsg"/>
      </sequence>
    </foreach>
  </sequence>
</process>
```

The workflow component provides functionalities for defining the control flow logic of business processes. Workflows are specified as XML scripts following the BPEL standard. The basic activities are the individual services. Services can be composed using the following constructs: sequence, switch, and foreach. We plan to add more features in the future, and the ultimate goal is to have a BPEL compatible workflow implementation. Figure 6 shows the definition of the *CreateIOI* workflow in our language. Each process handles one message, and a set of properties can be defined as XPATH queries on the message. The first receive construct initiates a new process, and populates the property fields in the message header. Conditions in switch can only be specified as predicates on the property fields. Instead of using a centralized workflow controller, a workflow specification is compiled into a set of routing rules and executed distributedly. The compilation is done by modeling a workflow as a finite state machine: states are service invocations and transitions are routing rules. The set of properties in the foreach statement is assigned to a special field in the header

"ReferenceProperites" as list of values, and the router creates a new instance of the message for each value in the list.

The transactional correctness is guaranteed in two levels: the service level and the business process level.

At the service level, each service can be viewed as a transaction. Each transaction involves the following steps:

- pickup a message from the queue.
- do some computations on the message.
- save the result in database.
- send one or more messages out.

and we need to guarantee that

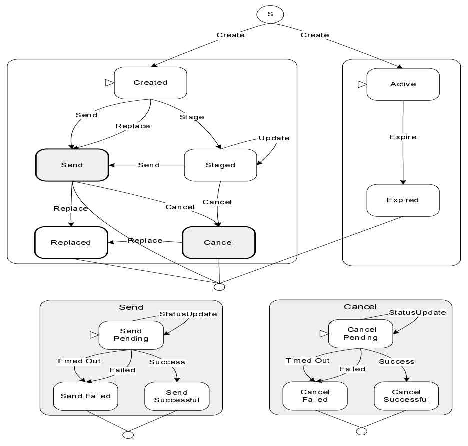
- messages are never lost and duplicated,
- messages in each service's incoming queue are processed one and only once, and
- the processing of the messages satisfies the ACID properties.

This can be achieved by employing a two-phase commitment transaction control mechanism between the transactional message queues and the database. However, the two-phase commitment protocol is expensive and inefficient. Therefore, we adopted an approach based on timestamp and snapshot to guarantee the transactional properties. We assume each message has a unique ID and timestamp. Only messages with a newer timestamp can be committed into the database. Thus, inconsistent and duplicate message processing is aborted. Messages are not removed from the incoming queue until they are processed. Thus, uncommitted messages will not be lost. The approach is sketch as in algorithm 3.3.

Algorithm 3.3 *Transactional message processing*

```
msg = IncomingQueue.ReadHead();
do computations on msg;
begin transaction
  assign a new timestamp to msg
  if timestamp is newest
      save msg in DB as ``committed''
  else abort
end transaction
IncomingQueue.Delete(msg)
send msg to target queues
update msg in DB as ``finished''
```

The correctness at the business process level is guaranteed by a state chart model of the business object. Any update on the object corresponds to a transition in the state chart diagram. An update is allowed only if the transition is enabled. Figure 7 is an example of the state chart diagram for IOI objects.

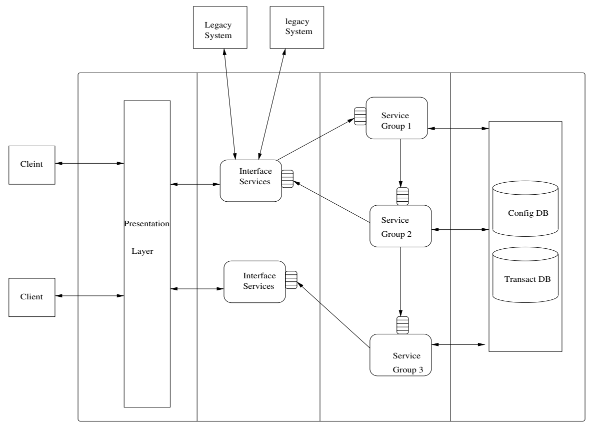


Resilience The system is comprised of loosely coupled services typically executing in a distributed enterprise environment, but providing transactional semantics. As such, no loss of messages can be tolerated. The system provides for message resiliency by the use of queues and durable queuing semantics. If a process that provides message routing (which enqueues messages) fails, the messages that it has yet to enqueue are retained in the corresponding server's disk storage system. When the message router process is restarted, it first processes any outstanding messages from disk, prior to handling new messages that are needed to be enqueued. Likewise, if a service invocation component fails, the state of its' queue is retained on disk. On process initialization, the service invocation component processes in order all messages existing in its' queue, including those retained prior to the process failure. To provide additional resilience in the case of disk failure, the queue state can be retained in highly reliable storage systems such as Redundant Array of Independent Disks (RAID) or Storage Access Networks (SAN) systems.

4 Implemenation Experience

We implemented the SOA framework using the Microsoft Windows 2003 and the .Net framework, and built a FIX based financial transactions system using the underlying SOA framework.

As shown in Figure 8, the system contains the following layers: the presentation layer, the interface service layer, the internal service layer, and the database layer. The presentation layer is set of web-based user interfaces for customers to manually manage the configuration of the system and perform transactions. The interface service layer exposes a set of Web service interfaces for clients to interact with the system automatically. In addition, the interface service layer contains hundreds of FIX endpoints that connects to



thousands of clients.

Services in the system are grouped into service groups. Each group of services is deployed in the same server. Resiliency and scalability are achieved by deploying the same group of services on a cluster of server boxes. We managed to scale our system to process 1000 transactions per second at periods of peak load, and millions of messages per day.

One of the challenges in our implementation is efficient message passing and transformation. To avoid unnecessary overheads of message transformation, messages are passed in their native format through message queues. However, when messages go through an interface service, it is transformed to a compatible format, in most cases, an XML document. The .Net framework provides several ways to handle an XML document. The first alternative is to use DOM. However, manipulating a DOM is inefficient. In our implementation, we use a customized object model and serialization/parsing methods to handle FIX messages.

System configuration during deployment can be a challenging problem because each service may require a different set of configuration parameters. To simplify the issue, we centralize the management of system configuration by using a single configuration database for the entire system. All servers bootstrap there configuration by loading the information from the configuration database on startup. To support the dynamic update of configuration information, when the configuration is updated, a message is sent to a notification service. The notification service then notifies each server regarding the configuration update.

When an error occurs during a transaction, details of the error are sent to an exception message queue. Messages in the queue are then reviewed by the administrator for appropriate handling. The system requires tractability of the history of all transactions, in case disputes or legal issues arise. The system provides an audit log service, which logs the history of all messages. The audit log can be reviewed, monitored, and queried.

5 Conclusions and Future Work

This paper proposed an SOA based framework for building high performance equity transaction systems. The framework supports interoperability with trading systems based on the FIX protocol, and also supports Web service standards. The system allows services to be configured and added flexibly. Business processes are specified as service compositions based on a workflow model. The system provides transactional correctness guarantees, reliability and fault tolerance. In the future, we plan to add more sophisticated workflow and routing features to the system, and to extend the framework to support more application domains in the financial services market, such as fixed income and foreign exchange.

References

[1] K. P. Birman. Like it or not, web services are distributed objects. *Commun. ACM*, 47(12):60–62, 2004.

[2] K. P. Birman, R. van Renesse, and W. Vogels. Adding high availability and autonomic behavior to web services. In *26th International Conference on Software Engineering (ICSE 2004)*, pages 17–26, 2004.

[3] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*, 2001. http://www.w3.org/TR/wsdl.

[4] F. Curbera, Y. Goland, J. Klein, F. Leymann, D. Roller, S. Thatte, and S. Weerawarana. *Business Process Execution Language for Web Services, Version 1.1*, 2003. http://www-106.ibm.com/developerworks/library/ws-bpel/.

[5] R. Fileto, L. Liu, C. Pu, E. D. Assad, and C. B. Medeiros. Poesia: An ontological workflow approach for composing web services in agriculture. *The VLDB Journal*, 12(4):352–367, 2003.

[6] fix.org. *The FIX Protocol*, 2002. http://www.fixprotocol.org/.

[7] M. Gudgin, M. Hadley, N. Mendelsohn, J.-J. Moreau, C. Henrik, and F. Nielsen. *SOAP Version 1.2 Part 1: Messaging Framework*, 2003. http://www.w3.org/TR/SOAP/.

[8] B. Medjahed, A. Bouguettaya, and A. K. Elmagarmid. Composing web services on the semantic web. *The VLDB Journal*, 12(4):333–351, 2003.

[9] OASIS. *Web Services Security (WS-Security)*, 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-secure/.

[10] OASIS. *Web Services Transactions specifications*, 2002. http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.

[11] Y. Petinot, C. L. Giles, V. Bhatnagar, P. B. Teregowda, H. Han, and I. Councill. A service-oriented architecture for digital libraries. In *ICSOC '04: Proceedings of the 2nd international conference on Service oriented computing*, pages 263–268. ACM Press, 2004.

[12] W. Vogels. Web services are not distributed objects. *IEEE Internet Computing*, 7(6):59–66, 2003.

[13] J. Zhang, J.-Y. Chung, and C. K. Chang. Migration to web services oriented architecture: a case study. In *SAC '04: Proceedings of the 2004 ACM symposium on Applied computing*, pages 1624–1628. ACM Press, 2004.

[14] O. Zimmermann, S. Milinski, M. Craes, and F. Oellermann. Second generation web services-oriented architecture in production in the finance industry. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 283–289. ACM Press, 2004.