Building Dependable and Secure Web Services

L. E. Moser,[1] P. M. Melliar-Smith,[1] and W. Zhao[2]

Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106 [1]
Department of Electrical and Computer Engineering, Cleveland State University, Cleveland, OH 44115 [2]

Email: {moser, melliar-smith}@ece.ucsb.edu, wenbing@ieee.org

***Abstract*— Web Services offer great promise for integrating and automating software applications within and between enterprises over the Internet. However, ensuring that Web Services can satisfy their clients' requests when their clients need them is a real challenge. In this paper we describe dependability technologies, including transparent SOAP connection failover, replication, checkpointing and message logging, in addition to reliable messaging and transaction management for which there exist Web Services specifications. We also present security technologies, including encryption and digital signatures for which Web Services specifications exist, as well as other security technologies. We discuss how these technologies can be applied to the components of a typical Web Services architecture to render business activities that span multiple enterprises dependable and secure.**

***Index Terms*— Availability, business activity, consistency, dependability, fault tolerance, intrusion tolerance, reliability, security, trust, Web Service**

I. Introduction

Web Services [9] enable the software of different enterprises to interact with each other, even if those enterprises use different hardware, different operating systems and different programming languages. Web Services can streamline business activities over the Internet by invoking operations automatically that, otherwise, would be invoked manually by a human through a browser and by enabling direct computer-to-computer interactions between the computers of different enterprises.

The potential widespread use and benefits of Web Services are compelling, because they facilitate:

- Automation of business activities distributed across multiple enterprises
- Collaboration among enterprises by coupling together the business processes running on their computer systems.

Web Services create opportunities for efficient ecosystems of consumers and suppliers, collaborating and competing for products and services over the Internet. Web Services can run not only on mainframe computers and server computers but also on client desktop computers and mobile handsets.

Figure 1 shows a simple example of a business activity that spans multiple enterprises and that consists of multiple Web Services. The customer (Company A) purchases a product from a distributor (Company B) using a Web Service. Company B, in turn, employs Web Services of three other companies, a supplier (Company C), a credit card company (Company D) and a shipper (Company E).

Despite their great promise, Web Services introduce new problems into enterprise computing, in particular:

- Faults or intrusions in the computer system of one enterprise can adversely affect another enterprise
- Data consistency, integrity and privacy are difficult to maintain
- Lack of availability, reliability and security can damage relationships between an enterprise and its customers, suppliers and partners.

These problems become more challenging as business activities become more automated, as Web Services trigger other Web Services, and as business activities involve more enterprises and more steps.

In this paper we discuss strategies and technologies for solving these and other problems that arise when implementing business activities as Web Services. We note that, as business activities increase in size, availability can deteriorate substantially unless mechanisms are employed to protect against faults. We also note that compensating transactions incur a greater risk of database inconsistency as business activities scale to large sizes. We review existing standards for Web Services related to dependability and security, and identify their strengths and weaknesses. We describe various technologies for increasing the dependability and security of Web Services, including transparent SOAP connection failover, replication, checkpointing, message logging, encryption and digital signatures. Finally, we show how to apply these technologies to a typical Web Services architecture.

II. The Need for Dependability and Security

A. Availability and Reliability

To ensure that Web Services can satisfy their clients' requests when their clients need them, all of the Web Services of a business activity, and all of the components

This journal paper is based on the conference paper "Making Web Services Dependable" by L. E. Moser, P. M. Melliar-Smith and W. Zhao, which appeared in the *Proceedings of the First IEEE International Conference on Availability, Reliability and Security*, Vienna, Austria (April 2006).

This research has been supported in part by MURI/AFOSR Grant F49620-00-1-0330 for L. E. Moser and P. M. Melliar-Smith at the University of California, Santa Barbara, and by a faculty startup award for W. Zhao at Cleveland State University.

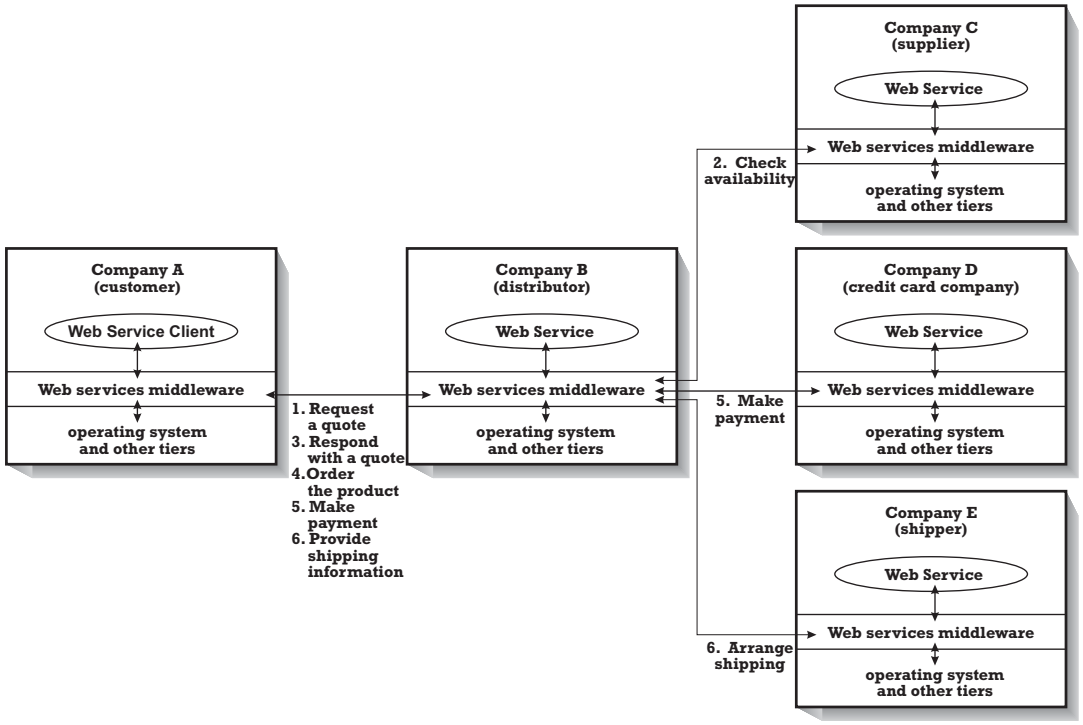

Figure 1. Use of Web Services in a business activity that spans multiple enterprises.

of those Web Services, must be dependable and secure. If one of the components of a Web Service is not available or reliable, all of the other components will be affected.

Availability is the probability that, if service is requested from a system, that service is provided. Informally, availability is a measure of the uptime of a system; five nines (0.99999) availability means that there can be at most 5.25 minutes of planned and unplanned downtime per year. *Reliability* is a measure of the time to occurence of a failure, including an incorrect result. In general, reliability is more difficult to ensure and to analyze than availability.

The availability of a business activity can be much less than the availability of any of the components of the Web Services that comprise that business activity, as the following example shows.

In this example, we let n be the number of tiers in a Web Services architecture within an enterprise and let m be the number of Web Services of different enterprises that are involved in a business activity. We assume that n is the same for all of the enterprises and that m is the same for all of the business activities. We assume further that the processes within the different tiers and within the different enterprises are independent.

We let p be the probability that the processes in any one of the tiers within an enterprise fails. Then, $1 - p$ is the probability that they do not fail. If all of the processes within those tiers are operational at the start of the business activity, then

$$q = (1 - p)^{mn}$$

is the probability that no fault occurs. For example, if $p = 0.00001$, $m = 4$ and $n = 3$, then the probability that no fault occurs is $q = (1 - p)^{12}$. The values of q for different values of $1 - p$ are shown in Figure 2.

For l independent business activities (*e.g.*, l business activities per day), the probability that no fault occurs in any of them is given by

$$r = q^l = (1 - p)^{lmn}$$

With the same values of m and n as above, *i.e.*, $m = 4$ and $n = 3$, and with $1-p = 0.99999$, the probability that no fault occurs is $r = (0.99999)^{12l}$. The values of r for different values of l are shown in Figure 2.

$m = 4, \quad n = 3$

1-p	q
0.9	0.282
0.99	0.886
0.999	0.9881
0.9999	0.9988
0.99999	0.99988

$1 - p = 0.99999$

l	r
10	0.99880
100	0.98807
1000	0.88692
10000	0.30119
100000	0.00001

Figure 2. The availability q of a single business activity based on the availability $1 - p$ of a single component, assuming $m = 4$ enterprises and $n = 3$ tiers, and the availability r of a number l of business activities for $1 - p = 0.99999$.

Many business computer systems must process 100,000 business activities per day, and the probability of complete success can be astonishingly low, as Figure 2 shows. Because of the nature of Web Services and business activities, all of the components of all of the Web Services involved in a business activity must be highly available

in order to achieve a high probability that all of the business activities will complete successfully. Even with careful programming and testing, it is unlikely that the probability of a fault in a step of a business activity will be reduced below 0.00001. Therefore, the required levels of availability cannot be achieved realistically without fault tolerance, recovery and retry, which must be regarded as essential to make business activities using Web Services available and reliable.

Data consistency is also crucial for Web Services, where a business activity spans multiple enterprises over the Internet and where detecting and correcting inconsistencies is difficult, time consuming and expensive.

Transactions have been used successfully to maintain data consistency within an enterprise. However, they have been used much less in wide-area distributed systems. One of the problems with distributed transactions based on the Two Phase Commit (2PC) protocol is that the participants incur the risk that their data will be locked, and will become inaccessible, for an arbitrarily long time, if the transaction coordinator fails. In theory, this risk can be mitigated by a Three Phase Commit (3PC) protocol or by replicating the transaction coordinator [24], [43]. In practice, few transaction processing systems use a 3PC protocol because of the high overheads in the fault-free case, and because replication of the transaction coordinator presents challenging problems as discussed below. Moreover, transaction commit does not scale well as the number of participants in the transaction increases.

Currently, business activities are typically implemented using multiple local transactions, with compensating transactions [11] to abort business activities that cannot be completed. Unfortunately, compensating transactions are difficult to design and program, have a high error rate, and incur a high risk of data inconsistency.

Figure 3 shows the probability of potential inconsistency for a business activity, for various rates of failure of individual local transactions, when using compensating transactions. Compensating transactions are assumed to incur the same fault rate as regular transactions, although realistically their fault rate is probably higher. Although the risk of data being locked for a substantial period of time (because the transaction coordinator failed) is unacceptable, the risk of data inconsistency resulting from the use of compensating transactions is even more unacceptable. Consequently, mechanisms that prevent both the locking of data by failed transactions, and potential inconsistency of data resulting from incorrect compensations, are essential for reliable operation of business activities.

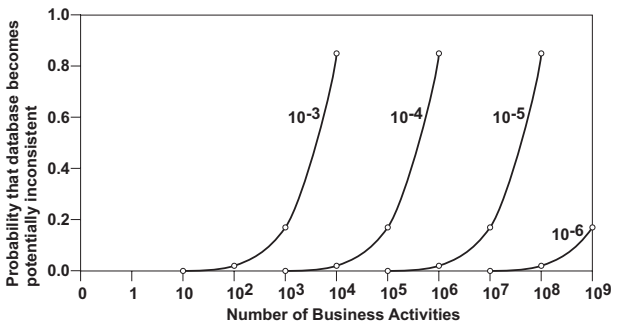

Figure 3. The probability that a database is left in a potentially inconsistent state after l business activities, for various rates of failure of individual local transactions, when using compensating transactions.

B. Security

Because Web Services operate in the open environment of the Internet, security is a critical concern for business activities implemented as Web Services.

Web Services expose interfaces and operations publicly, which presents potential security problems. Web Services use XML data formats with self-describing data elements, which reveal to hackers how to interact with them. Web Services are based on SOAP, which is based on HTTP, which allows messages to pass through enterprise firewalls. Web Services publish information about how to access and use their interfaces and operations in WSDL files, which facilitates and attracts attacks.

In a Web Services environment, sensitive and personal information (*e.g.*, credit card numbers) and business information (*e.g.*, customer contacts and employee names) are transmitted as parameters of the Web Services. Consequently, data privacy (confidentiality) and data integrity are important.

Data privacy ensures that the contents of the data transmitted between a sender and a receiver over the Internet are not viewed by a third party. *Data integrity* ensures that the data are neither damaged nor altered when they are in transit between the sender and the receiver.

Encryption is typically used to provide data privacy and data integrity for sensitive and personal information before it is transmitted over the network. However, once the data reaches the server, typically it is decrypted and stored in a single place, where a malicious user might be able to access it.

Authentication is used to verify that a person, a computer, or a computer program, such as a Web Service, is who or what it claims to be. Authentication can be used to provide:

- *Authorization*, which determines whether a privilege will be granted to a particular user
- *Non-repudiation*, which ensures that a user is not able to deny having done something that was authorized to be done.

In a Web Services environment, a Web Services provider might be authenticated by the Web Services client before the client invokes the Web Service and sends personal information. The client might also be authenticated by the Web Services provider before the provider executes the Web Service and returns critical information to the client in its reply.

Some clients might be authorized to invoke some operations of a Web Service, whereas other clients might be authorized to invoke other operations of the Web Service. A preferred client might have its own endpoint to the

Web Service, whereas less preferred clients might share a single endpoint.

Traditional methods for non-repudiation, such as digital signatures, are vulnerable to *forgery*. Digital transactions are also potentially subject to *fraud*, *e.g.*, when computer system is broken into or infected with viruses. Participants can claim such fraud to repudiate a transaction.

III. Background

A. Web Services

Web Services standards [9] define the syntax of Web Services documents, the format of messages, and the means to describe and find Web Services. They do not define implementation mechanisms or application program interfaces, which remain proprietary to individual vendors. Different vendors can implement Web Services infrastructures in different ways. Thus, Web Services standards provide interoperability between Web Services that are implemented using different hardware, different operating systems and different programming languages, but they do not provide portability of application programs from one platform to another. The basic Web Services standards comprise:

- The eXtensible Markup Language (XML), which defines the syntax of Web Services documents, so that the information in those documents is self-describing.
- The Simple Object Access Protocol (SOAP) for XML messaging and mapping of data types, so that applications can communicate with one another.
- The Web Services Description Language (WSDL) for describing a Web Service, its name, the operations that can be called on it, the parameters of those operations, and the location to which to send requests.
- The Universal Description Discovery and Integration (UDDI) standard, which is used by the Registry where providers publish and advertise their Web Services, and clients query and search for Web Services to discover what the services offer and how to access them.

B. Dependability and Security

In [3] Avizienis, Laprie, Randell and Landwehr have presented the basic concepts of, and a taxonomy of, dependable and secure computing. We highlight some of the key terminology and definitions below.

1) Replication: In dependable systems, replication protects a server application against faults, so that if one replica becomes faulty, another replica is available to provide the service to the clients. Replication is typically used for the *crash fault* model, which requires $2f + 1$ replicas to tolerate up to f faulty replicas. The most commonly used replication strategies are passive, active and semi-active replication, which are summarized below.

- *Passive replication*: There is a single primary replica that executes the operations invoked by the clients on the server application, and one or more backup replicas that do not execute those operations. The replication infrastructure transfers a checkpoint of the primary to the backups periodically or at the end of each remote invocation.
- *Active replication*: All of the replicas execute the operations invoked by the clients on the server application independently and at approximately, but not necessarily exactly, the same physical time. A checkpoint is used only to bring up a new active replica.
- *Semi-active replication*: Both the primary and the backup replicas execute each invoked operation. The primary provides directives (such as the order in which messages are to be processed) to the backups. The backups follow those directives and, thus, lag slightly behind the primary in executing the operations. Only the primary communicates results and invokes further operations.

2) Checkpointing: In dependable systems, checkpointing (*i.e.*, recording the state of a replica) is used by all replication strategies but in different ways. Passive replication uses checkpointing during normal operation. Active replication and semi-active replication do not use checkpointing during normal operation, but they do use it to initialize a new or recovering replica. The checkpoints of an application process can be stored on disk, or they can be transmitted to and stored on another processor.

If a fault occurs, the application process is restarted on the same or a different processor and its state is restored using the most recent checkpoint. There are two kinds of checkpointing, namely, application-aware checkpointing and application-transparent checkpointing.

- *Application-aware checkpointing*: The application programmer implements *getState()* and *setState()* methods. The *getState()* method captures particular parts of the application state and encodes that state into a byte sequence. The *setState()* method decodes the byte sequence and restores the application state from the checkpoint.
- *Application-transparent checkpointing*: The checkpointing infrastructure uses operating system mechanisms [23] to capture the state of the application process (including file descriptors, thread stacks, etc), without the need for the application programmer to implement the *getState()* and *setState()* methods.

There are two kinds of application-transparent checkpointing, namely, full checkpointing and incremental checkpointing:

- *Full checkpointing*: The checkpointing infrastructure captures the entire memory image of the application process (including file descriptors, thread stacks, etc), without knowing the data structures of the application program.
- *Incremental checkpointing*: The checkpointing infrastructure captures only those pages of the memory image that have changed since the last checkpoint. The infrastructure transmits the incremental check-

point to the designated processor or disk, and the incremental checkpoint is merged with the most recent full checkpoint stored there.

3) Encryption: To provide data privacy (confidentially), encyption is used to obscure information (*e.g.*, hide the contents of a message) so that it is unreadable without special knowledge. To encrypt a message, the sender can use a pre-established security key, or can generate a symmetric key on-the-fly and include the key (encrypted with the receiver's public key) in the message.

The Secure Sockets Layer (SSL) is the most common way of implementing data privacy. SSL provides public key encryption using a public key and a private key. It also supports authentication of the sender and, optionally, authentication of the receiver.

However, SSL provides only a point-to-point security solution. In a Web Services environment, where Web Services are chained together and multiple enterprises are involved, end-to-end security is required. Moreover, SSL incurs a high overhead because it encrypts and decrypts entire messages, when perhaps only some of the information in the message needs to be encrypted.

4) Digital Signatures and Message Authentication Codes: Each of these technologies is used to provide authentication.

Digital signatures are typically implemented using public key encryption. First, the sender produces a *digest* of the message by hashing the message using a secure hash function such as SHA1 [25] or MD5 [35]. Then, the sender generates a digital signature using the sender's private key. When the receiver receives the encrypted digested message, it decrypts the digest and transforms the digest back to the original message using the sender's public key. The receiver can then be certain that the owner of the private key signed the document. Using a message digest (rather than the entire message) reduces the cost of encryption and decryption.

For a digital signature to have validity, the receiver must be confident that the key is owned by the entity that the receiver thinks owns the key. To protect against an imposter interacting with the receiver, a certificate issued by a trusted Certificate Authority is often used to match a public key with the actual entity.

A *Message Authentication Code (MAC)* is a short piece of information that is used to authenticate a message. A MAC differs from a digital signature in that the sender and the receiver use a single secret key. The sender and receiver of a message must agree on the key before initiating communication. A MAC does not provide non-repudication like a digital signature does.

C. Web Services Dependability and Security Standards

The Web Services community has published several specifications related to reliable messaging, transaction management and security.

1) Reliable Messaging: The Web Services Reliable Messaging (WS-ReliableMessaging) specification [8] and the Web Services Reliability (WS-Reliability) specification [36] define application-level reliable messaging protocols that operate over SOAP.

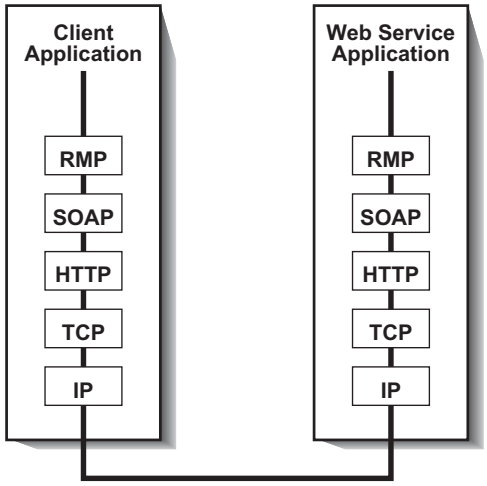

Figure 4. Reliable messaging protocol stack.

SOAP typically operates over HTTP, which in turn operates over TCP, which operates over IP. Even though TCP delivers messages reliably, and in order, to the Transport Layer, there is no guarantee that SOAP messages sent over HTTP are reliably delivered all the way up the protocol stack to the destination application. If a SOAP message is not delivered successfully (*e.g.*, because it has an incomplete address), the sender application receives a response containing a SOAP fault element that gives status or error information.

Both WS-Reliability and WS-ReliableMessaging provide reliable messaging for SOAP to the destination application, using acknowledgments and retransmissions with different quality of service levels, including at least once, at most once, exactly once and source ordered message delivery. Figure 4 shows the protocol stack, where RMP stands for Reliable Messaging Protocol.

2) Transactions and Business Activities: The Web Services specifications [10], [11], [12] for both short-running transactions and long-running business activities that span multiple enterprises aim to provide data consistency and protection against faults.

The Web Services Transaction (WS-Transaction) specification [10] includes protocols for atomic distributed transaction commitment, based on the Two Phase Commit (2PC) protocol. Transaction processing based on the 2PC protocol, as defined by the WS-Transaction specification, provides data consistency for Web Services applications. However, if the transaction coordinator fails and all of the participants in the transaction have voted to commit but have not received a commit from the coordinator, the 2PC protocol blocks and requires the participants to wait for the coordinator to recover, which can take an arbitrarily long time.

The Web Services Business Agreement (WS-Business Agreement) protocols [11] support long-running transac-

tions that span multiple enterprises, are not two-phase, and allow the business logic to determine whether the business activity should roll forward or roll backward.

The Web Services Coordination (WS-Coordination) specification [12] describes a framework for plugging in protocols that coordinate the actions of distributed applications, including those that require strict consistency and those that require agreement of only a proper subset of the participants. A Web Service creates a context that is used to propagate an activity to other Web Services and to register for a particular coordination protocol. Participants make heuristic decisions regarding the outcome of transactions. However, continued processing without waiting for coordinator recovery can compromise data consistency.

3) Security: The XML Encryption specification [41] builds on the Secure Sockets Layer (SSL) and provides an end-to-end solution that addresses the drawbacks of SSL. Using XML Encryption, a client can selectively encrypt an XML element or its contents. Moreover, a client can encrypt XML elements that are intended for different parties using different keys. To reduce the overhead of encryption and decryption, some XML elements might not be encrypted at all.

The XML Digital Signatures specification [40] provides of way to represent a digital signature in XML. With XML Digital Signatures, a client can sign different parts of a document using different digital signatures. Because the digital signatures are in-line, the signed documents can be easily archived and later retrieved.

The WS-Security specification [30] defines end-to-end security mechanisms for SOAP. WS-Security achieves *integrity* by attaching a digital signature of the sender to the message. Moreover, it achieves *privacy* (confidentiality) by encrypting all or part of the message, often the message body. In addition, it achieves *authentication* by embedding a security token in the message.

More specifically, WS-Security defines a set of SOAP extensions as XML structures that employ XML Encryption and XML Digital Signatures within the context of SOAP messages. WS-Security specifies XML structures for security claims and security tokens. A *security claim* is a statement that a client (or a user) makes, such as the client's name, identity, key, rights and privileges. A *security token* is a collection of security claims. The most simple form of security token is the username/password token, where the sender of the message specifies a username and a password (typically in digest form), and the receiver verifies it. WS-Security also defines a timestamp-based mechanism to defend against replay attacks.

Several other Web Services specifications extend WS-Security to include more comprehensive secure messaging support. The Web Services Trust (WS-Trust) specification [2] describes mechanisms for security token management (issuance, renewal, revocation and verification) and methods to establish trust relationships using the security tokens. The Web Services Federation (WS-Federation) specification [5] describes mechanisms that enable the sharing of user identity and authentication information across different trust domains, so that the target services do not require the users' local identities.

IV. Dependability and Security Technology

Dependability and security technology can increase the levels of availability, reliability, consistency and security for Web Services. This technology includes reliable messaging, replication, checkpointing and restoration, message logging and replay, and transactions, as well as security mechanisms, as discussed below (see also [22]).

Neither the WS-Reliability specification nor the WS-Reliable Messaging specification addresses the topics of message persistence and recovery from faults. However, these mechanisms are essential for reliable operation of business activities composed of one or more Web Services, and they are tightly coupled to the reliable handling of messages.

If a Web Service fails after a message has been delivered to it and after it has acknowledged receipt of the message but before it has fully processed the message (*e.g.*, because it has invoked a nested request), the following actions are required.

- The recovering Web Service must be restored to the checkpointed state it had at some moment preceding the fault.
- The TCP connections must be restored.
- Messages received subsequent to checkpointing the state must be replayed from a log on distributed or persistent storage.
- Messages, generated by the recovering Web Service, that have already been delivered to other Web Services, must be detected and suppressed.

A. Reliable Messaging

WS-Reliability and WS-ReliableMessaging can be readily extended to make the re-establishment of the connections of a Web Service transparent to remote clients and servers so that they do not need to reissue requests or replies. We refer to this capability as *transparent SOAP connection failover*.

Transparent SOAP connection failover involves the reliable message header and body for a group or sequence of messages and, in addition for WS-Reliability, the state that was negotiated for the agreement, before transmitting the messages in that set. The messages can be logged on disk for local restart, or in the volatile memory of a backup computer for failover to the backup computer.

B. Replication

Replication is used in fault-tolerant systems to protect a server application against faults, so that if one replica becomes faulty, another replica is available to provide the service to the clients. As discussed previously, there are several different replication strategies, namely, active, passive and semi-active replication.

The most challenging aspect of replication is maintaining the consistency of the replicas, as operations are invoked on the service, as the states of the replicas change, and as faults occur. Replica consistency is obviously critical for active and semi-active replication, which must maintain the consistency of two or more concurrently executing replicas. Less obviously, replica consistency is also important for passive replication because a recovering replica must repeat computations and communications with other Web Services since the most recent chechpoint. Those computations and communications must be consistent with the prior computations and communications to avoid disrupting other Web Services. Maintaining replica consistency requires the sanitization of non-deterministic operations and also the handling of side-effects, as discussed below.

C. Checkpointing and Restoration

Checkpointing is used by all replication strategies but in different ways. Passive replication uses checkpointing during normal operation. Active replication and semi-active replication do not use checkpointing during normal operation, but use it to initialize a new or recovering replica.

The checkpoints of an application process can be stored on disk, or can be transmitted to and stored on another processor. If a fault occurs, the application process is then restarted on the same or a different processor, and the most recent checkpoint is used to restore the process to the state it had at the time of the checkpoint.

For applications that involve multiple threads within a process or data structures that contain pointers, it is difficult to implement the *getState()* and *setState()* methods of application-aware checkpointing. On the other hand, application-transparent checkpointing does not produce checkpoints that are portable across hardware architectures, because a checkpoint taken as a binary image contains values of variables that differ for different architectures, such as memory addresses.

D. Message Logging and Replay

For simple restart without replication or checkpointing, and for all replication and checkpointing strategies, an infrastructure that provides dependability must support message logging and replay. Again, the messages can be logged either in the memory of another processor or on disk. However, logging the messages on disk and subsequently replaying them from disk can adversely affect performance.

For restart without either replication or checkpointing, the entire message history (from the first message to the most recent message in the set) must be retained and all of the messages must be replayed to the new or recovering replica. For replication and checkpointing, only the messages since the most recent checkpoint need to be replayed.

E. Sanitizing Non-Deterministic Operations

Extensive research has been undertaken on the topic of sanitizing non-deterministic operations (see, *e.g.*, our previous work [31], [42]).

Messaging is one source of replica non-determinism, because messages can be received by the replicas in different orders, due to loss of messages and retransmissions, delays in the network, etc. To maintain replica consistency, messages must be delivered to all of the replicas reliably and in the same order. Such a message delivery service is called *atomic broadcast* or *atomic multicast* [16]. For passive replication, the infrastructure must log messages on disk or in the memory of another processor so that, if the primary replica fails, the messages after the checkpoint can be replayed. For both active and passive replication, the infrastructure must detect and suppress duplicate invocations and duplicate responses.

Another source of replica non-determinism is multithreading. If two threads within a replica share data, they must claim and release mutexes to protect that shared data. However, the threads in different replicas will likely run at slightly different speeds and, thus, they might claim mutexes in different orders. To maintain replica consistency, the mutexes must be granted to the threads within the replicas in the same order.

Other sources of replica non-determinism include operating system functions that return values local to the processor on which they are executed, such as *rand()* and *gettimeofday()*, or inputs for the replicas from different redundant sources, or system exceptions due to, say, exhaustion of memory on one of the processors. Such sources of replica non-determinism must be sanitized, so that all of the replicas see the same values of the functions, the same inputs from the redundant sources, and the same system exceptions. This sanitization must be done, regardless of which replication strategy is used.

F. Handling Side-Effects

In addition to sanitizing non-deterministic operations, side-effects that occur as the result of a client's invoking operations on a Web Service must be handled properly to achieve replica consistency.

In particular, if a Web Service writes data to a file or a database, those operations must be handled correctly. The actions taken depend on whether each replica has its own copy of the database or the file, or the replicas have a single shared copy. Similarly, if a Web Service sends messages to, or invokes operations on, other Web Services, those operations can have side-effects that must be handled properly.

G. Transactions and Business Activities

Within a single enterprise, transactions have been successfully used to provide data consistency and to protect data against faults by means of their ACID properties, as discussed previously.

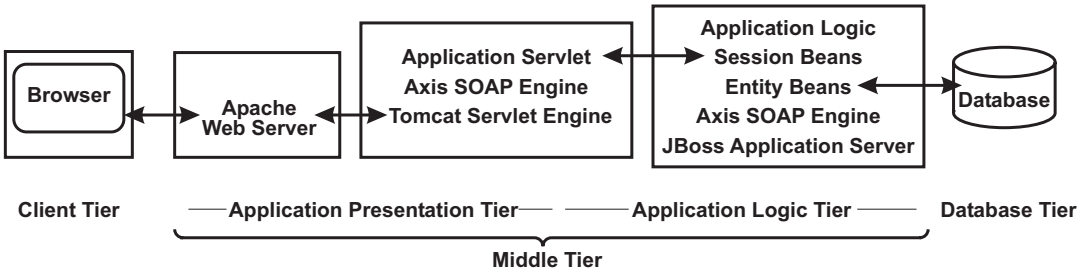

Figure 5. Three-tier Web Services architecture.

It is possible to achieve higher availability of Web Services that employ transactions by using transactions and replication together. In particular, by replicating the transaction coordinator, the 2PC protocol can be rendered non-blocking and exactly-once semantics can be provided for the clients' invocations. Moreover, by replicating the middle-tier components and using transparent transaction retry, roll-forward recovery can be achieved. We have implemented an infrastructure for CORBA [43] that replicates the transaction coordinator and also the middle-tier application objects to protect the business logic processing and to avoid potentially long service disruptions caused by failure of the coordinator. A similar infrastructure for Web Services can provide high availability, reliability and data consistency.

H. Security Mechanisms

As described previously, WS-Security achieves message authentication by embedding a security token in the message. The most simple form of security token is the username/password token. For better protection, more sophisticated security tokens such as the SAML token [6], Kerberos token [19] or X.509 token [20] can be used.

For many kinds of Web Services, the dependability mechanisms and the secure messaging mechanisms described above are sufficient. However, some mission-critical Web Services, such as those used in financial transactions (*e.g.*, banking and stock trading) require a higher degree of dependability and security, which can be achieved by means of more sophisticated and stringent mechanisms to handle, for example, intrusion attacks [39].

Instead of trying to prevent every single intrusion, *intrusion tolerance* allows and tolerates them. Intrusion tolerance is related to *Byzantine fault tolerance* [28], which is used for the arbitrary fault model that requires $3f + 1$ replicas to tolerate up to f faulty replicas. Like conventional fault tolerance, the most effective approach for achieving intrusion tolerance is to distribute the processing and storage across a network of computers. The mechanisms that ensure totally ordered message delivery under the Byzantine fault model involve three phases of message exchange between the replicas. The cost is significantly higher than that of conventional fault tolerance.

Confidentiality (privacy) in the presence of intrusions can also be realized through distribution. The concern for confidentiality requires mechanisms beyond Byzantine fault tolerance, because naive replication of secret data across several sites is perceived to reduce confidentiality (there are more sites available for penetration to gain access to a secet). One promising approach is the separation of secret processing from normal processing [45]. Secret processing is carried out using a secret sharing scheme such as the Fragmentation-Redundancy-Scattering (FRS) method [18], and normal processing is replicated using Byzantine fault tolerance mechanisms.

V. Dependable and Secure Web Services Architectures

Web Services applications are usually programmed in Java or .Net. We refer here to a typical Java-based implementation.

A three-tier Web Services architecture involves clients in the first tier, a Web server, a servlet engine and/or a J2EE application server in the middle tier, and a database system in the third tier, as shown in Figure 5. The clients communicate with the Web server and invoke operations of the Web Service, which is deployed in a server-side container. The server-side container can be a servlet container such as Tomcat, or a container in a J2EE application server such as JBoss or Geronimo. The Axis SOAP engine works with both types of containers. Typically, there are multiple clients and multiple Web servers that are used for load balancing the clients' requests.

In a typical Web Services use case, the client accesses a Web page, which consists of HTML and Java Servlets or JSP scripts. The client can make invocations by clicking on one or more links provided in the Web page. Once a request reaches the servlet engine, a servlet creates dynamic content for the Web page, and might also perform simple business logic processing and communicate with the database system. Applications that involve complex business logic processing typically use a J2EE application server. Multiple J2EE application servers might be used to load balance the clients' requests. The servlet creates dynamic content for the presentation of the Web page and communicates with the container, which contains session beans that perform the business logic processing and entity beans that correspond to the database records.

Within an enterprise, the components of a Web Service can be made dependable and secure using the technology discussed previously, as shown in Figures 6, 7 and 8 and discussed below.

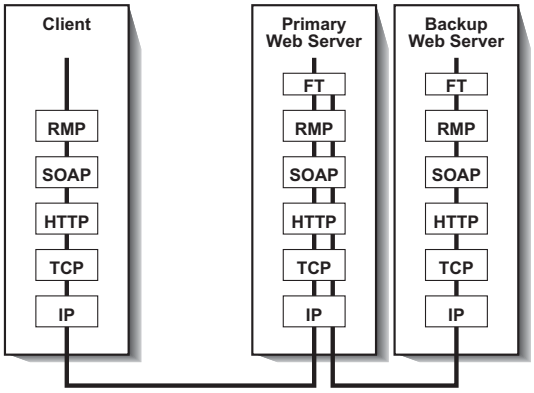

Figure 6. Fault-tolerant Web server.

A. Web Server

A Web server is sometimes regarded as "stateless" in that it does not retain any application state between its response to a client's request and the client's next request. The application state is either stored in the database or returned to the client in a URL or a cookie. However, during the processing of a client's request, the Web server does maintain application state and also hidden internal state, such as the progress of nested invocations or disk accesses or the state of the connections with the servlet engine or the database. The state that results from actions that are visible to other processes must also be captured, and restored if a fault occurs.

If the Web server fails while processing a client's request, either the unreplicated Web server must be restarted and the client must reissue the request, or the replicated Web server must be failed over from the primary to a backup on another processor, as shown in Figure 6. In the latter case, the fault tolerance infrastructure must replay the client's requests to the restarted or backup Web server after the checkpoint has been restored. In either case, the Web server must not write the state to the database, or send a response to the client, more than once.

In addition, the infrastructure at the servlet engine or the J2EE application server must ensure that the restarted or backup Web server receives its response messages reliably and in the correct order, if the Web server fails.

B. Servlet

To achieve high availability, fault tolerance must be provided for the containers and the servlets contained within them using replication, checkpointing and message logging, as shown in Figure 7.

Even if the state of the servlet application, such as the state stored in a session object, is written to the database before the servlet sends a response back to the Web server, a checkpoint must be taken that includes the state within the Tomcat containers, such as the connections with the Web server and the J2EE application server or database server. Subsequently, if the servlet engine fails while it is processing a client's request and interacting with the J2EE application server or the database server, the servlet engine must be brought back up or failed over to a backup servlet engine, its state must be restored from the checkpoint, and the messages after the checkpoint must be replayed.

C. J2EE Application Server

Some three-tier Web Services architectures use J2EE application servers. The J2EE standard derives from the CORBA standard, and mandates the use of CORBA's Internet Inter-ORB Protocol (IIOP). The CORBA Object Transaction Service [33] provides data consistency through atomic commitment of distributed transactions. The Fault Tolerant CORBA standard [32] provides high availability by replicating the application objects. Several implementations of Fault Tolerant CORBA exist (see, *e.g.*, [31], [42]).

Fault tolerance must be provided for the J2EE containers and the beans contained within them, as shown in Figure 7, even if the application is coded as one or more transactions with roll backward recovery and the beans are entity beans whose states are stored in a database. Again, the reason is that the J2EE container contains considerable hidden internal state. If application-transparent checkpointing is used, the interactions and overlap between the checkpoint of the J2EE container process and the states of the entity beans stored in the database must be reconciled.

D. Transaction Coordinator

To achieve both high availability and data consistency for transaction-based applications, the transaction coordinator must be replicated. Replication of the transaction coordinator renders the 2PC protocol non-blocking and achieves exactly-once semantics for the clients' invocations [24]. Furthermore, if the middle-tier servers are also replicated and transparent transaction retry is used, roll-forward recovery can be achieved [43].

E. Database Server

Much work has been done on improving the reliability and availability of database systems. Vaysburd [38] has provided an excellent survey of commercial packages that provide fault tolerance for database systems, with respect to such requirements as persistence, data consistency and availability of service.

F. UDDI Registry

The UDDI Registry that contains the WSDL descriptions is critically important in providing dependability for Web Services. If a Web Service fails, a client can query the Registry to obtain the latest information on the Web Service. If the latest binding information is different from the binding information used by the failed Web Service, a client can retry the Web Service with the new binding. Similarly, if the original Web Service invocation exceeded a timeout or the timeout changed, a client can retry the

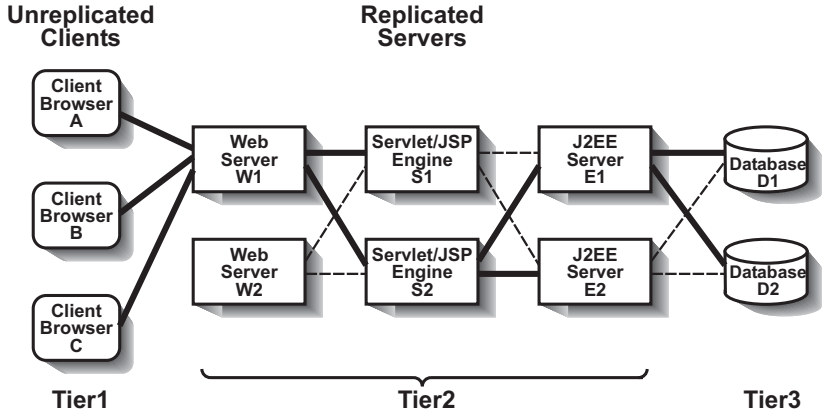

Figure 7. Fault-tolerant three-tier Web Services architecture.

Web Service with the new timeout. If the retry is not successful, a client can use the Registry to select another provider of a similar Web Service.

Consequently, the UDDI Registry must be readily available for the clients that are looking for the WSDL descriptions of the Web Services that they need. Availability of the UDDI Registry can be achieved using the replication, checkpointing and message logging technology, described above. Realizing that the availability of the UDDI Registry is critically important, OASIS has published a specification for lazy replication of the UDDI Registry [15], where the updates are propagated point-to-point from one replica to another replica.

G. Business Activities

Distributed transactions based on the 2PC protocol are seldom used for business activities that span multiple enterprises, because they unavoidably involve one enterprise's locking the data records of another enterprise. Instead, extended transactions, as defined by the WS-BusinessActivity [11] and WS-Coordination [12] specifications, are used. Extended transactions typically involve local transactions and compensating transactions [11] that offset committed local transactions when a business activity is rolled back. However, compensating transactions can have undesirable effects, such as one transaction's seeing the results of another transaction before the compensating transaction is applied.

In [44] we presented a reservation-based extended transaction protocol for Web Services that coordinates business activities and that avoids the use of compensating transactions. Each task within a business activity is executed as two steps. The first step involves an explicit reservation of resources according to the business logic. The second step involves the confirmation or cancellation of the reservation. Each step is executed as a separate traditional transaction.

H. Proxy Server Architecture

In the proxy server architecture, shown in Figure 8, a proxy server acts as an intermediary between the client and the Web Services. The proxy server accepts incoming calls from the client and invokes one or more Web Services to fulfill the client's request. If one of those Web Services is not available, the proxy server invokes an alternate Web Service that provides a similar service.

The proxy server can itself be a Web Service that wraps or invokes the underlying Web Services. Alternatively, it can be a servlet that accepts HTML or XML data, parses the data, invokes the required Web Services, and returns the results to the client as HTML or XML.

The proxy server architecture is particularly useful for mobile clients, where the communication between the mobile client and the proxy server is over a cellular or wireless network, and the communication between the proxy server and the Web Services is over a wired network. By not using SOAP or XML between the mobile client and the proxy server, the power consumed by the mobile client, the latency seen by the client, and the bandwidth used by the client are reduced.

I. Security Strategies

As discussed previously, security requires an end-to-end solution, which involves not only technologies but operational processes. The entire path that the data takes is important, not only the exchange of data between the client and the Web Service.

Encryption is used to ensure confidentiality (privacy) of a client's sensitive and personal information when it is transferred across the network as parameters of a Web Service. When the encrypted data reaches the server, it is decrypted. Some businesses store the data in-the-clear within a flat text file, and perhaps back up the data on another disk drive. They might also forward the data in-the-clear to another business or to a human.

Several different strategies exist for making this data handling and forwarding more secure. One strategy is for the server to encrypt the information before it stores or forwards the data. Another strategy is for the client to encrypt the data in such a way that the receiving server can decrypt only part of it. Partial decryption, as is done in XML Encryption, allows the receiving server to read only

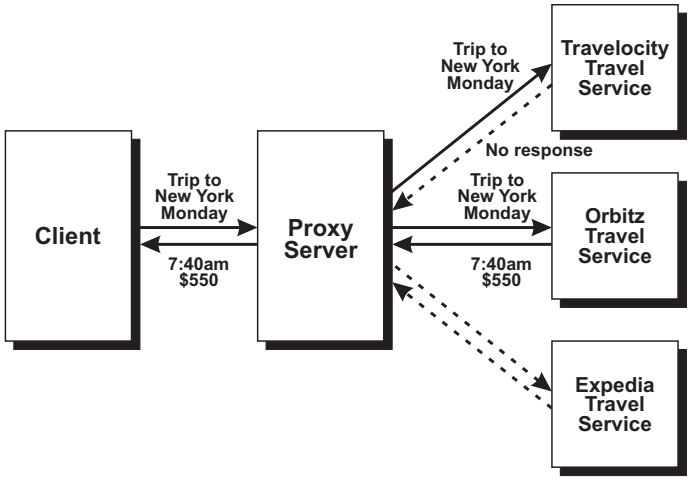

Figure 8. The proxy server architecture.

certain information, such as routing information, so that the rest of the information can be forwarded as encrypted data to the appropriate party.

Authentication of Web Services can be provided using several different strategies, which can be categorized as system-level and application-level strategies.

System-level strategies do not require custom application programming of the Web Service or changes to the Web Service if the authentication strategy is changed. The operating system, the middleware or the Web server handles the authentication and authorization prior to forwarding the SOAP request to the Web Service. System-level strategies are typically based on credentials (usernames/passwords) or digital certificates.

Application-level strategies require custom development, and modification when the authentication mechanism changes. Application-level strategies are based on credentials included in the SOAP message header or body. If the credentials are part of the SOAP message body, the invoked Web Service must parse the credentials, as well as implement the authentication and authorization mechanisms. If the credentials are part of the SOAP message header, a service other than the invoked Web Service can parse and authorize the invocation.

VI. Related Work

As discussed previously, there are a number of Web Services standards that relate to dependability and security, some of which have been implemented in widely deployed Web Services middleware such as WebLogic, WebSphere, JBoss and Apache.

In [34] Pallickara, Fox and Pallickara have provided an excellent analysis of the WS-Reliability and WS-ReliableMessaging specifications. They identify the similarities and differences of the two specifications, and recommend extensions to the protocols to ensure ordered delivery across sets of messages and across multiple destinations. They also discuss how the two specifications might be used together. We have discussed deficiencies of those specifications and have described mechanisms that can be used to overcome them.

Aghdaie and Tamir [1] have investigated failover of connections for Web servers (not Web Services) and replay of messages for Web servers up to the HTTP layer of the protocol stack, by modifying the Linux kernel and the Apache Web server. Their work is similar to our transparent TCP connection failover [26], which is intended for general kinds of applications, including Web Services. However, implementing failover support in the operating system kernel is not portable across different operating systems.

Several researchers have undertaken extensive research on the topic of sanitizing non-deterministic operations in order to maintain replica consistency (see, *e.g.*, our previous work [31], [42]). In particular, messages must be delivered to the replicas reliably and in the same order, which is called atomic broadcast or atomic multicast [16]. Recognizing this need, the Java Messaging Service (JMS) has been extended to provide atomic multicast [27].

Hanik [21] has described in-memory session replication for the Tomcat servlet engine that uses the Java Groups group communication toolkit. His strategy exploits Java's serializability to checkpoint and restore session state, but restricts what can be put into a session object to ensure serializability. Moreover, his approach stores request data in decentralized sources, such as temporary files, which can result in data inconsistencies.

Bartoli, Prica and di Muro [7] have presented a framework for program-to-program interaction across unreliable networks and an implementation in a Tomcat servlet container. Their prototype is based on replication of HTTP client session data and replication of a counter. It provides the same consistency guarantees as a non-replicated service with respect to the order of execution requests. Moreover, it ensures that, even if a client issues duplicate requests (*e.g.*, because of a fault), the service executes the client's request at most once.

Babaoglu, Bartoli, Maverick, Patarin and Wu [4] have described a framework for prototyping J2EE replication algorithms. They divide the replication code into two layers, the framework which is common to all replication algorithms and a specific replication algorithm which is plugged into the framework.

Sun, Lin and Kemme [37] have implemented the OASIS lazy replication strategy for the UDDI Registry, mentioned previously, as well as an eager replication strategy. The eager replication strategy is based on middleware that employs a multicast group communication protocol. They provide response time, propagation time and execution results for both lazy and eager replication in a LAN and in a WAN.

Intrusion tolerance has been an active research topic in recent years [13], [14], [18], [39], [45]. However, research on intrusion-tolerant Web Services is rare. The most relevant work is that of Merideth, *et al.* [29], who have implemented a Byzantine fault-tolerant infrastructure for Web Services applications. Their implementation is based on the C++ library for Byzantine fault tolerance described in [14], which uses a proprietary UDP-based communication protocol. The major advantages of Web Services are interoperability and extensibility; the use of a proprietary messaging protocol negates both of them. Moreover, no attempt is made to tackle the issue of confidentiality.

VII. Conclusion

If Web Services are to achieve their objective of automating business activities across multiple enterprises, they must be made dependable and secure. The existing Web Services reliable messaging, transactions, business activity and security specifications must be augmented with additional mechanisms to provide higher levels of dependability and security.

In this paper we have described various technologies for increasing the dependability and security of Web Services, including transparent SOAP connection failover, replication, checkpointing, message logging, encryption and digital signatures. We have also shown how to apply these technologies to a typical Web Services architecture.

Some of the dependability and security technologies discussed in this paper incur high computation and communication overheads and, thus, are not ready for practical use. Moreover, there exist other dependability and security issues that need to be addressed, such as how to preserve confidentiality in the case of intrusion attacks.

Acknowledgment

The authors wish to thank the reviewers for their valuable comments.

References

[1] N. Aghdaie and Y. Tamir, "Implementation and evaluation of transparent fault-tolerant Web Service with kernel-level support," *Proceedings of the IEEE International Conference on Computer Communications and Networks*, Miami, FL, October 2002, 63-68.

[2] S. Anderson, *et al.*, "Web Services trust language," version 1.1, February 2005, http://www-128.ibm.com/developerworks/library/specification/ws-trust/.

[3] A. Avizienis, J. C. Laprie, B. Randell and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing* 1, 1, January-March 2004, 11-33.

[4] O. Babaoglu, A. Bartoli, V. Maverick, S. Patarin and H. Wu, "A framework for prototyping J2EE replication algorithms," *Proceedings of the International Symposium on Distributed Objects and Applications*, Agia Napa, Cyprus, October 2004, 1413-1426.

[5] S. Bajaj, *et al.*, "Web Services federation language," version 1.0, July 2003, http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnglobspec/html/ws-federation.asp.

[6] A. Barbir, M. Gudgin, M. McIntosh and K. S. Morrison, "SAML token profile," version 1.0, January 2006, http://www.ws-i.org/Profiles/SAMLTokenProfile-1.0.html.

[7] A. Bartoli, M. Prica and E. A. di Muro, "A replication framework for program-to-program interaction across unreliable networks and its implementation in a servlet container," *Concurrency and Computation: Practice and Experience* 18, 7, June 2006, 701-724.

[8] R. Bilorusets, *et al.*, "Web Services reliable messaging," February 2005, http://www-128.ibm.com/developerworks/webservices/library/ws-rm/.

[9] D. Booth, H. Hass, F. McCabe, E. Newcomer, M. Champion, C. Ferris and D. Orchard, "Web Services architecture," February 2004, http://www.w3.org/TR/ws-arch.

[10] L. F. Cabrera, *et al.*, "Web Services transaction," August 2002, http://www.ibm.com/developerworks/library/ws-transpec/.

[11] L. F. Cabrera, *et al.*, "Web Services business activity framework," January 2004, http://www.ibm.com/developerworks/library/ws-busact/.

[12] L. F. Cabrera, *et al.*, "Web Services coordination," September 2003, http://www.ibm.com/developerworks/library/ws-coor/.

[13] C. Cachin and J. Poritz, "Secure intrusion-tolerant replication on the Internet," *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Washington, D.C., June 2002, 167-176.

[14] M. Castro and B. Liskov, "Practical Byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems* 20, 4, 2002, 398-461.

[15] L. Clement, *et al.*, "UDDI replication specification," version 2.03, July 2002, http://uddi.org/pubs/Replication-V2.03-Published-20020719.pdf.

[16] X. Defago, A. Schiper and P. Urban, "Total order broadcast and multicast algorithms: Taxonomy and survey," *Computing Surveys* 36, 4, December 2004, 372-421.

[17] Y. Deswarte, N. Abghour, V. Nicomette and D. Powell, "An intrusion-tolerant authorization scheme for Internet applications," *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, Washington, D.C., June 2002, C-1.1 - C-1.6.

[18] Y. Deswarte, L. Blain and J. Fabre, "Intrusion tolerance in distributed computing systems," *Proceedings of the IEEE Symposium on Research in Security and Privacy*, Oakland, CA, 1991, 110-121.

[19] P. Griffin, C. Kaler, P. Hallam-Baker and R. Monzillo, "Web services security Kerberos token profile 1.1," OASIS specification, February 2006.

[20] P. Hallam-Baker, C. Kaler, R. Monzillo and A. Nadalin, "Web services security X.509 certificate token profile," OASIS specification 200401, March 2004.

[21] F. Hanik, "In-memory session replication with Tomcat 4," April 2002, http://www.TheServerSide.com.

[22] D. Ingham, S. Shrivastava and F. Panzieri, "Constructing dependable Web Services," *IEEE Internet Computing* 4, 1, January/February 2000, 25-33.

[23] G. Janakiraman, J, Santos, D. Subhraveti and Y. Turner, "Cruz: Application-transparent distributed checkpoint-restart on standard operating systems," *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, Yokohama, Japan, June 2005, 260-269.

[24] R. Jimenez-Peris, M. Patino-Martinez, G. Alonso and S. Arevalo, "A low-latency non-blocking commit service," *Proceedings of the International Conference on Distributed Computing*, Lisbon, Portugal, October 2001, 93-107.

[25] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, September 2001, http://www.faqs.org/fcs/rfc3174.html.

[26] R. Koch, S. Hortikar, L. E. Moser and P. M. Melliar-Smith, "Transparent TCP connection failover," *Proceedings of the IEEE International Conference on Dependable Systems and Networks*, San Francisco, CA, June 2003, 383-392.

[27] A. Kupsys, S. Pleisch, A. Schiper and M. Wiesmann, "Towards JMS compliant group communication," *Proceedings of the IEEE International Symposium on Network Computing and Applications*, Cambridge, MA, August 2004, 131-140.

[28] L. Lamport, R. Shostak and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems* 4, 3, 1982, 382-401.

[29] M. G. Merideth, A. Iyengar, T. Mikalsen, S. Tai, I. Rouvellou and P. Narasimhan, "Thema: Byzantine-fault-tolerant middleware for Web service applications," *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems*, Orlando, FL, October 2005, 131-140.

[30] A. Nadalin, C. Kaler, P. Hallam-Baker and R. Monzillo, "Web services security: SOAP message security 1.0," OASIS specification 200401, March 2004.

[31] P. Narasimhan, L. E. Moser and P. M. Melliar-Smith, "Strongly consistent replication and recovery of fault-tolerant CORBA applications," *Computer Science and Engineering Journal* 17, 2, March 2002, 103-114.

[32] Object Management Group, "Fault Tolerant CORBA," OMG Technical Committee Document formal/02-06-59, Chapter 23, CORBA/IIOP 3.0, 2000, http://www.omg.org.

[33] Object Management Group, "Transaction Service Specification," v1.2, OMG Technical Committee Document ptc/2000-11-07, 2000, http://www.omg.org.

[34] S. Pallickara, G. Fox and S. L. Pallickara, "An analysis of reliable delivery specifications for Web Services," *Proceedings of the IEEE Conference on Information Technology*, Las Vegas, NV, April 2005, 360-365.

[35] R. Rivest, "The MD5 message-digest algorithm," RFC 1321, April 1992, http://www.faqs.org/rfcs/rfc1321.html.

[36] T. Rutt, M. Peel, D. Bunting, K. Iwasa and J. Durand, "Web Services reliability," August 2004, http://oasis-open.org/committees/ tc_home.php?wg_abbrev=wsrm.

[37] C. Sun, Y. Lin and B. Kemme, "Comparison of UDDI registry replication strategies," *Proceedings of the IEEE International Conference on Web Services*, San Diego, CA, July 2004, 218-225.

[38] A. Vaysburd, "Fault tolerance in three-tier applications: Focusing on the database tier," *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, Lausanne, Switzerland, October 1999, 322-327.

[39] P. Verissimo, N. Neves and M. Correia, "Intrusion-tolerant architectures: Concepts and design," *Lecture Notes in Computer Science* 2677, 2003, 90-109.

[40] W3C, "XML signature syntax and processing, February 2002, http://www.w3.org/TR/xmldsig-core/.

[41] W3C, "XML encryption syntax and processing," December 2002, http://www.w3.org/TR/xmlenc-core/.

[42] W. Zhao, L. E. Moser and P. M. Melliar-Smith, "Design and implementation of a pluggable fault tolerant CORBA infrastructure," *Cluster Computing, Special Issue on Dependable Distributed Systems* 7, 4, October 2004, 317-330.

[43] W. Zhao, L. E. Moser and P. M. Melliar-Smith, "Unification of transactions and replication in three-tier architectures based on CORBA," *IEEE Transactions on Dependable and Secure Computing* 2, 1, January-March 2005, 20-33.

[44] W. Zhao, L. E. Moser and P. M. Melliar-Smith, "A reservation-based coordination protocol for business activities" *Proceedings of the IEEE International Conference on Web Services*, Orlando, FL, July 2005, 49-56.

[45] L. Zhou, F. Schneider and R. van Renesse, "COCA: A secure distributed online certification authority," *ACM Transactions on Computer Systems* 20, 4, 2002, 329-368.

ABOUT THE AUTHORS

Louise E. Moser is a professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. At SRI International, she was a key contributor to the design verification of the Software-Implemented Fault-Tolerant (SIFT) reliable aircraft control computer. Dr. Moser has served as an associate editor for *IEEE Transactions on Computers* and an area editor for *IEEE Computer* magazine in the area of networks and on numerous conference program committees. She is an active participant in various standards organizations, including the Service Availability Forum, where she was an editor of the Application Interface Specification. Her research interests include distributed systems, computer networks and fault tolerance. Dr. Moser has served as principal investigator for many R&D projects, and is the author or coauthor of more than 230 publications. She received a Ph.D. in Mathematics from the University of Wisconsin, Madison.

P. M. Michael Melliar-Smith is a professor in the Department of Electrical and Computer Engineering at the University of California, Santa Barbara. At GEC Computers in England, Dr. Melliar-Smith was principal designer of the GEC 4080, which won the Queen's Award for Innovation. At the University of Newcastle upon Tyne, he invented the definitions of fault, error and failure, as well as the recovery block method for software fault tolerance. As Senior Computer Scientist and Program Director at Stanford Research Institute, he was involved in the design of the Software-Implemented Fault-Tolerant (SIFT) aircraft flight control computer. Dr. Melliar-Smith has served as principal investigator for many R&D projects, and has authored or coauthored more than 250 publications. His research interests span the areas of distributed systems, communication networks and protocols, and fault tolerance. He received a Ph.D. in Computer Science from the University of Cambridge, England.

Wenbing Zhao is an assistant professor in the Department of Electrical and Computer Engineering at Cleveland State University. His research interests span the fields of fault tolerance, security, computer networks and distributed systems. He has authored or coauthored more than 40 conference and journal publications. One of his papers won the best paper award in the International Symposium on Performance Evaluation of Computer and Telecommunication System in 2002. Dr. Zhao received the Ph.D. in Electrical and Computer Engineering from the University of California, Santa Barbara.