Increasing Reliability of Web Services

Cezar Toader*

**"Politehnica" University of Timişoara, Automation and Applied Informatics Department, Romania (Tel: +40-256-403000; e-mail: cezar.toader@gmail.com).*

Abstract: The main subject in this paper is the concept of reliable Web services, which are the blocks for building large distributed systems. This paper presents a fault tolerant Web services architecture which maintain client transparency. Important functionalities such as replication, fault management and client transparency are analyzed.

Keywords: Web services architecture, reliability, replication, performance, relative delay.

1. INTRODUCTION

The Web is increasingly used for critical distributed systems, applications and services. Service-Oriented Architecture (SOA) using Web Services is accepted as the architecture capable to interconnect applications running on different operating systems and facilitate complex interactions between autonomous and heterogeneous systems both within organizations and between them.

Web Services 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 different enterprises.

The benefits of Web Services are significant because they facilitate automation of business activities distributed over the Internet across multiple enterprises and collaboration among enterprises by coupling together the business processes running on their systems.

Failures in Web applications can lead to erroneous processing or even crashes in important systems like ecommerce, banking or stock trading.

One of the key causes of service disruptions is server failure. Hence, fault tolerance techniques that allow providers to deliver uninterrupted Internet services despite server failures are increasingly important.

For many distributed systems based on Web services there is a large base of installed client-applications and it is very complicated to require all of those to be modified. This is the reason why service providers look for those fault-tolerance schemes that are client-transparent, which means they can operate without requiring any special action by the client and without modifying the client application. This client transparency is an important requirement with respect to both the client application and the client operating system.

In this paper we propose a fault tolerant architecture containing several servers grouped in one autonomous unit based on servers and Web services. This unit accepts connections from clients, receives their requests, executes a suite of internal operations, internally deals with operation faults when they appear, and sends a reply to the client according to its request.

This server group has important functionalities: *replication management, fault management, logging and synchronization, client transparency*.

The proposed architecture extends the Web Services architecture by adding new components in order to build a reliable autonomous unit: WS-Manager, WS-Status Logger, Request Logger, and Message Router.

The major contribution of this paper is the extension of Web Services architecture in order to increase system reliability and maintain client transparency.

The rest of the paper is structured as follows. Section 2 provides a short background on Web Services. Section 3 describes important aspects concerning concepts as dependability, system reliability, system failures, faults, fault-tolerance, and client transparency. Section 4 presents the proposed architecture and Section 5 describes implementation aspects and a performance evaluation. Section 6 presents related works and Section 7 presents final conclusions.

2. WEB SERVICES

Web Services standards 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 vendors (Booth, et al., 2004).

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.

3. DEPENDABILITY OF WEB SERVICES

In complex applications Web services need to connect to other Web services in order to form composite Web services and complex Service Oriented Architectures (SOA). If one component in this chain of services is not available or reliable the whole system is affected.

A correct service is delivered when it implements the system function. A system failure occurs when the delivered service is different from the correct service. This deviation means that the service does not comply with its well-defined specification. An error is that part of the system state that may cause a failure. A fault is the cause of error. A fault may be active or dormant. When is active, the fault produces an error and, subsequently, a system failure (Avizienis, et al., 2004).

Web Services introduce new problems into enterprise computing, in particular:

- Faults in the computer system of one company can adversely affect another company;
- Data consistency, integrity and privacy are difficult to maintain;
- Lack of availability, reliability and security can damage relationships between a company 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 (Moser, et al., 2007).

Dependability is an integrative concept that encompasses several attributes: availability, reliability, safety, confidentiality, integrity, maintainability (Avizienis, et al., 2001). *Availability* means "the readiness for correct service" whereas *reliability* means "continuity of correct service".

One important mean to attain dependability is fault-tolerance. This term means "how to deliver correct service in the presence of faults".

In so-called *dependable systems*, replication is widely accepted technique to avoid system failures. Thus, system architects implement a service using a group of redundant, physically independent, servers, so that if some of these fail, the remaining ones still have the capability to offer the service to clients (Cristian, 1991).

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. The most commonly used replication strategies are passive, active and semi-active replication (Moser, et al., 2007).

4. THE PROPOSED FAULT-TOLERANT ARCHITECTURE

This section presents the proposed Web Services architecture enhanced with fault-tolerance.

We propose a fault tolerant architecture containing several servers grouped in one autonomous unit based on servers and Web services. This unit accepts requests from clients over a functional connection and sends replies to the clients according to their requests. This group of servers and Web services internally deals with operation faults when they appear and maintain client transparency.

This unit is an "autonomous best-effort delivery system" according to the client's requests. The autonomy means that, over a functional connection, the client application is not aware of internal system faults and recovery.

The analysis starts with the well-known three-tier architecture, as shown in Figure 1.

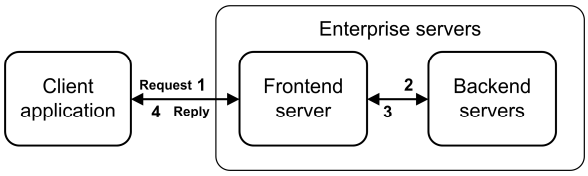

Fig. 1. The three-tier system architecture.
(The reply 4 will be sent to the client after steps 2 and 3.)

The *goal* is this: using a functional network connection between the client application and the enterprise's frontend servers, the client must receive a reply from the enterprise application according to the request previously made and all service faults which may appear within enterprise LAN must be internally solved and must not be shown to the client.

In order to achieve this goal, important functionalities must be implemented on the server side: *replication management*, *fault management*, *request logging*, *service synchronization*, and *client transparency*.

The proposed architecture contains specific components implementing the above-mentioned functionalities.

4.1 System components

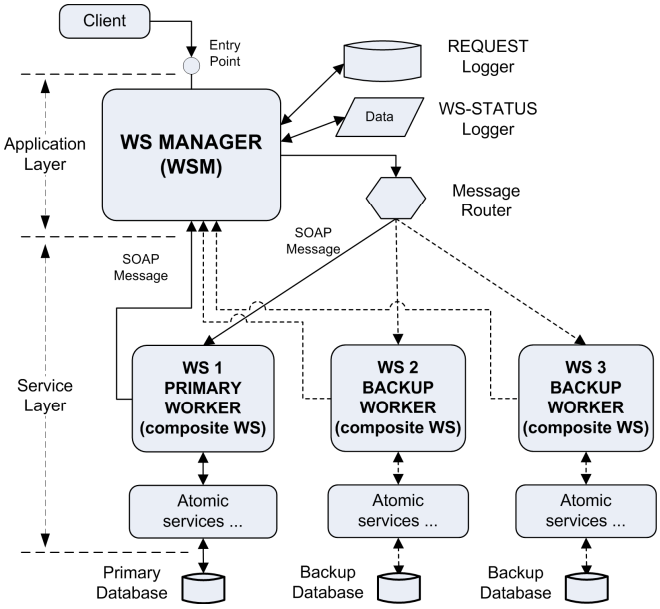

Fig. 2. The proposed architecture.

The client requests are received by Web Services Manager (WSM) at the Entry Point, as in Figure 2.

WSM is responsible for fault detection and recovery after service faults. In order to do this recovery, WSM uses a Request Logger which saves in a local store all client requests in a well defined order. Every request gets an ID in order to uniquely identify it afterwards. The stored messages will be used when a subsystem must be updated after a crash.

WSM acts like a monitor of web services. It maintains a data structure called WS-STATUS where important information about the service status is written.

After checking the web services status, WSM acts like a dispatcher. It uses a Message Router to redirect the current request to a functional Web service called *primary worker* or to a functional Web service called *backup worker*, when the primary worker status is FAULT.

The proposed architecture has two layers:

- *Application Layer*, which contains WS Manager, Request Logger, WS-STATUS Logger and Message Router;
- *Service Layer*, which contains many composite Web Services (WS 1, WS 2 ... WS n), and Atomic Services.

4.2 Assumptions

We start based on several *assumptions*:

- a functional network connection is established between the client and WSM;
- all client requests arrives at the Entry Point (EP);
- faults may appear within the system at the level of backend services;
- An ID is given by WSM to each request;
- WSM locally stores all requests including their ID;
- WSM is able to detect primary worker faults and redirect the request to backup worker(s);
- The stored list of requests is used when a worker is back online after a fault.
- The client application discovered the service by using a Service Registry (UDDI) and its request will be processed within the system.

4.3 Normal operation

When a request is arrived at Entry Point (EP), in the first step, WSM creates a Request ID for easy identify this message and stores both the Request ID and the message content in its own local database. This information can be retrieved afterwards, if necessary. Thus, WSM implements an important functionality: *request logging*.

Table 1. Logging requests

Request ID	Message content
RQ20100315-0000001	... (XML code ...)
RQ20100315-0000002	... (XML code ...)
...	...

WSM maintains a data structure called WS-STATUS. At a given time, a working Web service can be only in one of two states: READY or BUSY. After the initialization phase of the system, every worker is READY. This status is written in the structure WS-STATUS. A WS worker is given a job when its state is READY. When a worker starts a job, WSM notes its state as BUSY. Also, WSM notes the start time for that job in WS-STATUS. When the job is done and the worker is READY again the execution time is written in WS-STATUS.

The normal fault-free operation is done by WSM and primary worker as follows. WSM reads the information within WS-STATUS data structure to check the primary worker's state. This state could be READY, BUSY, or FAULT.

In Table 2, the following situation occurred: WS Manager needs to send a specific Request to all services, but their status is different.

The status of WS 3 is FAULT, so the job cannot start. The same job was started on WS1 at a specific time, it was successfully done in 38 milliseconds and the current state of WS1 is READY. The job was started on WS2 at a specific moment, but WS2 didn't finish the job. Its status is BUSY. Afterwards, WS2 may finish the job successfully and return to the status READY. The execution time will be written.

Table 2. WS-STATUS

WS	RQ ID	STATUS	Starting time	Execution time (ms)
WS1	RQ...010	READY	18:21:15,652	38
WS2	RQ...010	BUSY	18:22:35,083	N/A
WS3	RQ...010	FAULT		N/A

At any time, one of the services is chosen as Primary Worker. At the beginning, the primary worker is WS1.

If the state of primary worker WS1 is READY, then the following operations are executed:

1) WSM uses the Message Router to send the request to the Primary Worker WS1.
2) WS1 sends an acknowledgment to WSM
3) WSM writes in WS-STATUS the new state of Primary Worker (BUSY) and the starting time.
4) WS1 solve the request and sends the reply to WSM.
5) WSM receives data and writes the new state of WS1 (READY) and the execution time in WS-STATUS.
6) WSM sends the reply to the client.

If the state is BUSY then WSM waits for a specific amount of time and try again until the primary worker's state becomes READY, or until the maximum number or retries is reached. The case of fault in the system is described below.

4.4 Faults, recovery, and synchronization

From its internal database, WSM is able to find which was the last successfully Request ID on every WS worker.

There are a variety of situations where errors can occur in the primary worker's normal operation. The taxonomy of faults is presented by Avizienis, et al., (2004).

A fault occurs when the execution of the worker is required but results are not returned. Due to the nature of the distributed system, faults can be situated in different places:

- in the primary worker's host;
- in the primary worker's process on the host;
- in the execution thread on the host;
- at the level of atomic services;
- on the connection between worker's host and WSM;
- on the connection between worker's host and the host of atomic services and/or database systems.

In time, faults can occur:

- during request transmission to the primary worker;
- during processing of the request by the worker;
- during processing in the atomic services;
- during the process of sending replies from database systems to atomic services;
- during the process of sending replies from atomic services to primary worker;
- during the process of sending replies from worker to the WS Manager.

All the above-mentioned situations could determine the primary worker to remain indefinitely in the state BUSY or even the worker's process to crash.

If the status of primary worker is BUSY (not ready), but the backup worker is READY, then the above-mentioned steps (from 1 to 6) are realized by that backup worker. During this time, a *recovery from error* must be started for primary worker. WSM is responsible for sending all missed requests to the primary worker. All missed requests will be processed in the same order as they arrived in order that both services WS1 and WS2 to be in the same final state. Thus, WSM implements another required functionality: *service synchronization*.

All requests involving data writing in databases must be processed in a transactional way. If an operation succeeded, then WSM will ask Request Logger to write this in local database containing all requests.

Logging all requests, working with databases in a transactional way, and writing the success of the requests are very important elements in case WSM experiences a crash. After WSM restarting, the newly created process will retransmit to workers all logged requests which don't have the attribute *successfully done*. This mechanism allows the *recovery of WS Manager from failure*.

All these faults, recovery operations and service synchronizations are *client transparent.* The client application is not aware of all these operations. The system faults are masked by using a group of web services (workers) and a manager able to deal with fault detection, replication management, logging and recovery.

4.5 Changing the Primary Worker

WSM maintains statistics about workers activity in its own database: total working time during last 5 minutes, and last 30 minutes, and, very important, the ID of the last successfully solved request. Hence, according to its configuration, WSM is able to detect which was the fastest worker in the last 5 minutes or in the last half hour.

Periodically, according to its configuration, WSM will choose the fastest WS as the new Primary Worker, or it will maintain the current one.

5. IMPLEMENTATION DETAILS

Our implementation used identical systems for hosting server processes. The configuration is indicated below, in Table 3.

The Web server was IIS 7, running on a Windows Server 2008 operating system, with .NET Framework 3.5 SP1 installed. MS SQL Server 2008 was used as database server.

Table 3. Server configuration

OS	Windows Server 2008 Standard
Framework	.NET Framework 3.5 SP1 with WCF
DB server	Microsoft(R) SQL Server 2008
Web Server	Internet Information Server IIS 7
IDE	Visual Studio 2008 Professional
CPU	Intel(R) Core 2 Quad, Q6600, 2.4 GHz
Memory	2 GB, DDR2
Network	Intel(R) 82566DC, Gigabit Network
Hard disk	SATA, WD2500AAKS, 7200 RPM

The necessary WCF (Windows Communication Foundation) services were created. Then an ASP.NET application was built as hosting environment for these services. The working network segment was separated from the rest of the LAN to avoid any unnecessary network connection.

There are several distinct operations performed within the system. Some of them could be considered *major operations* unlike others called *preparatory operations*.

We consider as major operations the following two: primary worker processes the client request (including here all subordinated calls to databases and other services), and the response message is sent by primary worker to WSM. Their duration is noted t_{PROC}, respectively t_{RESP}.

We consider as preparatory operations the following: the client request is logged by the Request Logger, then the request is forwarded to Primary Worker which sends an ACK message to WSM which writes in WS-STATUS the new state of worker as BUSY. The duration is noted t_{PREP}.

The preparatory operations occur before the major operations. We consider faults may appear during the main operations and, as a result, their period is affected by the appearance of faults and the recovery process.

In order to evaluate the performance degradation when replication process occurs we modified the initial code to report the time of starting processing a request and the time of reply. Several hardware failures were simulated in order to see if the system is able to recover. We calculate the time span for different type of operations.

A small delay appears when the replication occurs. A more visible delay occurs when the system executes services synchronization. The absolute value of these delays is not so relevant because the size of the requests and especially of responses may vary on a large scale.

A more appropriate indicator of replication influence is the *relative delay of the system*, defined as:

$$RD = \frac{t_r}{t_0} \tag{1}$$

Here t_0 is the time span of a certain operation without replication, and t_r is the time span of the same operation when replication occurs.

According to previously made notations, the normal operation time span is as follows:

$$t_0 = t_{PREP} + t_{PROC} + t_{RESP} \tag{2}$$

When a fault occurs and the system realizes a *recovery from error*, the operation time span depends on partial processing (t_{P_PROC}) and recovery duration ($t_{RECOVERY}$) as follows:

$$t_r = t_{PREP} + \left(t_{P\_PROC} + t_{BLK} + t_{RECOVERY}\right) + t_{RESP} \tag{3}$$

During our experiments, the values for relative delay were significantly influenced by the size of SOAP message, the blocking duration t_{BLK}, and recovery duration as follows:

$$RD = 1.12 \div 1.43 \tag{4}$$

The model analyzed here treats requests one after another, no matter of the client who send it. In our experiments we did not consider any form of client prioritization. More clients simply mean more requests, but the operation mode remains.

If the number of clients increases, the system will be increasingly loaded and the response time seen by the client will also grow. In this paper we discussed about fault-tolerance and recovery from errors. According to the principle of separation of concerns, the overload issues are analyzed and solved by different load-balancing techniques.

Our experiments were based on intentionally caused faults. We searched for a proof of concept. In the real systems, the fault appearance and the nature of the faults are not easy to predict. Specific tests are necessary on a real system in a real environment. Only then engineers can determine significant parameters such as number of failures for the worker during the last 24 hours, or total system delay during last 24 hours. Apart from fault-tolerance and system recovery, further experiments concerning load-balancing in a multi-client environment could be very useful for dependable systems.

6. RELATED WORKS

Web services enable application-to-application interaction built on top of Web protocols. Moser, et al, (2007) presented an overview of techniques for building dependable and secure Web services.

The availability of network services may be increased by using many schemes providing a system to route new requests that arrive after a fault to a working server. But such schemes do not support recovery of in-progress requests. The published approaches include the use of DNS system to remove the address of faulty servers from service (Brisco, 1995), and schemes which direct clients to an alternate server replica (Suryanarayanan & Christensen, 2000)

There are various fault tolerance schemes for network services that are not client-transparent. These are so-called *client-aware solutions*. Some of them require modifications to the client application while others only require changes to the client OS kernel (the TCP implementation). One class of client-aware solutions are implementations of a 2-phase commit protocol on a 3-tier system, ensuring the transactions are performed exactly once and that *in-progress* requests are recovered (Frolund & Guerraoui, 1999).

The Web services community has developed application-level reliable messaging protocols built on top of SOAP and HTTP: WS-Reliable Messaging 2009 (Davis et al. 2009). But these protocols are not client-transparent and do not address some key topics such as message persistence and recovery from fault (Moser, et al., 2007).

A framework for building fault-tolerant Web services on top of SOAP is FT-SOAP. It facilitates a configuration with a primary and a warm backup replica, where a replication manager is able to promote the backup to be the new primary. But the client has to be modified in order to be able to redirect a request to the backup replica (Fang et al. 2007).

7. CONCLUSIONS

In this paper, a fault tolerant architecture was conceptualized. The proposed two-layer architecture internally deals with faults, without modifying interoperability of existing enterprise services. Internal algorithm in this architecture allows developers to use different programming models for Web services.

Important functionalities of this reliable architecture were analyzed in the paper: replication, fault management, logging, recovery, and client transparency.

The client applications, such as Web browsers, and client operating systems are not under the control of service providers. Hence, relying on client-side participation for fault tolerance is not practical and was not discussed here.

The proposed architecture is based on a Web Services Manager, a Request Logger, and a Status Logger, working together. Based on information given by the loggers, the Web Services Manager is able to decide which Web Service is the primary worker and when the system synchronization must be done. The client application sees this system as an autonomous entity giving the appropriate responses and internally dealing with faults.

Implementation aspects and experimental details and results were presented. An evaluation of the influence of replication phase on overall system performance shows acceptable delays in case of system synchronization.

REFERENCES

Avizienis, A., Laprie, J.C., and Randell, B. (2001). *Fundamental Concepts of Dependability*, Research Report no. 1145, LAAS-CNRS, 2001.

Avizienis, A., Laprie, J.C., Randell, B. and Landwehr, C. (2004). Basic concepts and taxonomy of dependable and secure computing, In *IEEE Transactions on Dependable and Secure Computing*, vol.1, no.1, 2004, pp. 11-33.

Booth, D., Hass, H., McCabe, F., Newcomer, E., Champion, M., Ferris, C., and Orchard, D. (2004). Web Services architecture, Available at: http://www.w3.org/TR/ws-arch [Accesed 30 March 2010].

Brisco, T. (1995). DNS support for load balancing, RFC 1794, Available at http://www.faqs.org/rfcs/rfc1794.html [Accessed 30 March 2010].

Cristian, F. (1991). Understanding fault-tolerant distributed systems, In *Communications of the ACM*, vol.34, Issue 2, 1991, pp. 56-78.

Davis, D., Karmarkar, A., Pilz, G., Winkler, S., and Yalcinalp, U. (2009). Web services reliable messaging (WS-Reliable Messaging) version 1.2, 2009, Available at http://docs.oasis-open.org/ws-rx/wsrm/200702 [Accesed 30 March 2010].

Fang, C.L., Liang, D., Lin, F., and Lin, C.C (2007). Fault tolerant Web services, In *Journal of Systems Architecture* no.53, Issue 1, January 2007, pp. 21–38.

Frolund, S., and Guerraoui, R. (1999). CORBA fault-tolerance: why it does not add up, In *The Seventh IEEE Workshop on Future Trends of Distributed Systems*, Tunisia, South Africa, 1999, pp. 229.

Moser, L.E., Melliar-Smith, P.M., and Zhao, W. (2001). Building dependable and Secure Web Services, In *Journal of Software*, vol.2, no.1, Febr. 2007, pp. 14-26.

Suryanarayanan, K., and Christensen, K. (2000). Performance evaluation of new methods of automatic redirection for load balancing of Apache Web servers distributed in the Internet, In *IEEE 25th Conference on Local Computer Networks*, 2000, pp. 644–651.