Toward a Modular and Efficient Distribution for Web Service Handlers

Beytullah Yildiz[1], Geoffrey C. Fox[2]

[1]Department of Computer Engineering, TOBB University of Economics and Technology, Ankara, Turkey
[2]Department of Computer Science, Indiana University, Indiana, USA

Abstract

Over the last few decades, distributed systems have demonstrated architectural evolvement. One recent evolutionary step is Service-Oriented Architecture (SOA). The SOA model is perfectly engendered in Web services, which provide software platforms to build applications as services. Web services utilize supportive capabilities such as security, reliability, and monitoring. These capabilities are typically provisioned as handlers, which incrementally add new features. Even though handlers are very important, the method of utilization is crucial for attaining potential benefits. Every attempt to support a service with an additional handler increases the chance of an overwhelmingly crowded handler chain. Moreover, a handler may become a bottleneck due to its comparably higher processing time. In this paper, we present Distributed Handler Architecture (DHArch) to provide an efficient, scalable, and modular architecture. The performance and scalability benchmarks show that the distributed and parallel handler executions are very promising for suitable handler configurations. The paper is concluded with remarks on the fundamentals of a promising computing environment for Web service handlers.

Keywords: Service-Oriented Architecture; Web service; parallel computing; pipelining; Web service handler

1. Introduction

One recent evolutionary step in the computing environment is Service-Oriented Architecture (SOA). Its goal is to achieve loosely coupled, scalable, and interoperable software systems. Many efforts have been made thus far. Remote Procedure Call (RPC), Remote Method Invocation (RMI), Common Object Request Broker Architecture (CORBA), and the Distributed Component Object Model (DCOM) were all intended to offer a promising distributed environment. However, for increased quality, new ideas and technologies have started emerging. SOA has been introduced to define the necessities and requirements for a solution. It has given birth to a new technology: Web services.

The Web service framework offers standard ways to interoperate between software applications running on a variety of platforms [1]. It provides seamless and loosely coupled communications. Many specifications have been introduced so far, and many others are on the way. The key features of Web services, which are described by the World Wide Web Consortium (W3C), have been presented as WS-specifications. Simple Object Access Protocol (SOAP) [2], Web Service Description Language (WSDL) [3], and Universal Description Discovery and Integration (UDDI) [4] are the most popular specifications.

The most famous specification among them is SOAP, which offers message-level agreement and is perfectly suited for the exchange of information. Agreeing on the message format on the wire and leaving the processing method to the interacting nodes increases the loose coupling feature. Moreover, SOAP has an extensibility feature. By utilizing that, the Web service container, or the middleware, in other words, provides an environment with additive and supportive functionalities and capabilities, called Web service handlers.

Handlers offer new capabilities or functionalities, such as security, reliability, monitoring, and so on, without increasing the complexity of service. Simplicity is a very crucial feature of an application. In the Web service structure, simplicity originates from a very well-known notion, *partitioning.* A whole task is divided between the handlers and the service endpoint. Instead of a large, hardly manageable application, clearly separable smaller tasks are more plausible. Charles Antony Richard Hoare states this very essential feature for designing excellent software in *The Emperor's Old Clothes* [5]. He says: "There are two ways of constructing a software design. One is to make it so simple that there are obviously no deficiencies; the other is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult."

Handlers are a very crucial component of Web services because of their key importance for execution. However, the utilization of handlers and their structures become important when the number of the necessary additive functionalities increases. The efficiency becomes essential when power-hungry and time-consuming

functionalities are introduced into the execution path. For instance, reliability adds a significant amount of processing time. Similarly, security may necessitate powerful machines to conclude tasks within a reasonable amount of time. Any additional handler can worsen the response time of a service. On the other hand, a service cannot be banned from obtaining new features. It is predestined that services will necessitate new capabilities to present a better computing environment. In other words, services eventually attain more functionality and capability in their execution paths. Accordingly, we may wind up with an overwhelmingly crowded handler pipeline, which makes the services slower. In other words, a Web service becomes *fat*; while the service is acquiring new capabilities, the response time becomes longer and management of the service becomes more difficult. Secondly, a handler may cause a *convoy effect*. In an execution pipeline, a handler may delay the service processing due to the fact that its execution is too slow. In other words, the handler becomes a *bottleneck*. This condition increases the number of request messages waiting to be served every second.

However, networks are becoming faster. Machines are becoming more powerful and their speed is constantly improving. Bottlenecks can be eradicated by delivering some of the handlers to a more powerful computing environment. This distribution reduces the burden on a single computing node. Additionally, application parallelism has been utilized for decades. Hence, handlers can be executed concurrently. The parallelism boosts the performance and provides a very effective and powerful solution. Moreover, multi-core processors are being widely utilized; even personal computers leverage cores, offering the opportunity for parallel executions and contributing to the parallel handler execution even without the introduction of any network latency.

Although the distribution of the handlers is very crucial in improving efficiency and scalability, there are other requirements to be able to benefit from the distribution. Running handlers concurrently necessitates additional structures and requirements. Hence, in this paper, we investigate the distribution of Web service handlers and their parallel execution. We also elaborate on the required support structures to understand this execution environment. Section 2 presents conventional Web service handler structures and explains how they utilize handlers. We elaborate on Distributed Handler Architecture in section 3. The performance and scalability benchmark results and analysis are provided in section 4. Finally, we conclude our findings in section 5.

2. Conventional Web service handler structures

There are several conventional Web service handler structures that facilitate the adding of new capabilities to Web services. Apache Axis [6] is currently the most dominant container in the Web service community and has a plethora of applications developed around it. There are two main versions, Apache Axis 1.x and Apache Axis 2. Apache Axis 1.x facilitates the incremental addition of capabilities to the Web service endpoint by leveraging handlers. Handlers can be in the request path and/or the response path. There exist two types of handlers. The first type contains singleton handlers, which do not require a peer. They can be deployed on either the client or server side. On the other hand, there are handlers that do require peers on the client or the service sides. For instance, a compression handler, which reduces the size of messages on the client side, requires an inverse handler on the service side, which performs the appropriate decompression.

Apache Axis 2 has more extensive and modular architecture. The core modules are separable from the remaining modules, so new modules can be added on the top of the core modules [7]. To handle information and keep the states, Apache Axis 2 defines an information module, which has a hierarchical structure that helps manage the object lifecycles. Apache Axis 2 basically views every transaction as a single SOAP processing. A top layered framework is necessary to implement a complex SOAP messaging, containing several messages. The Apache Axis 2 framework contains two pipes: IN and OUT. These may be combined to exchange messages. User applications can create a SOAP request by using a client API. Before handing the message over to the transport sender, new capabilities can be added with the handlers. Additionally, Apache Axis 2 introduces an upper level abstraction on top of the handler layer: a module. A module may contain a set of handlers and phase rules. In other words, it groups a set of handlers to provide a specific functionality.

Similar to Apache Axis, Web Service Enhancements (WSE) from Microsoft support Web services by offering an environment for the capabilities, which are called *filters*. The execution structure of the filters is very similar to that in Apache Axis 1.x. Both output and input filters are capable of processing SOAP messages. Although WSE has several built-in filters, customizable filters can be added, too. Filters can also create a chain of handlers.

Finally, DEN/XSUL provides a structure that offers an environment for the handler execution. The XML Services Utility Library (XSUL) is a modular Java library to construct Web and Grid services [8,9]. It provides a framework for XML-based processing and supports doc-literal, request-response, and one-way messaging. Furthermore, it contains modules for a lightweight XML/HTTP invoker and processor. DEN addresses the

performance and scalability bottleneck. It targets the Web service security processing steps directly, without touching the endpoint service logic at all. It granulates the application and makes the pieces separate processing nodes. These nodes are distributed across the Grid.

3. Distributing Web service handlers

The conventional handler structures do not utilize distributed computing very well. The execution is mainly performed in a single memory space. Moreover, handler level parallelism is not benefited. Although some structures make use of pipelining to run multiple instances of a handler chain in a single memory space and others touch the parallel execution in a restricted domain for security, distributed computing and parallelism are not fully utilized. Therefore, in order to gain the full benefit from Web services for the handler execution, we designed Distributed Handler Architecture (DHArch), a framework that provides functionalities for processing handlers concurrently and sequentially in the distributed environment. The goal is to remove the boundaries that keep the handlers in a single memory space and to contribute to modularity, reusability, interoperability, scalability, and responsiveness.

Although DHArch offers additional resources and more powerful computing environments via distributed computing, it also provides environments for parallel execution. An ideal parallel computation is one that utilizes completely independent programs such that there is no communication between them. This is known as *embarrassingly parallel* [10]. Each program receives different or the same data and creates an output without any input from the other programs that are running simultaneously. This type of execution requires a manager that distributes the data to the programs and collects and combines the results. This structure is very appropriate for the *message-passing programming* model.

Embarrassingly parallel programming uses *partitioning*, which is the basis of all parallel programming. It is a technique that divides the problem into parts. When we look at a Web service, we can see a main task and various supportive tasks. In order to distribute and parallelize them, first of all, *partitioning* needs to be applied. Many supportive tasks, handlers, are completely independent from each other. In other words, there is not any communication between them. Hence, it is perfectly reasonable for DHArch to benefit from embarrassingly parallel programming.

When we look at the distribution of a handler, the cost would be:

$$W = T_{comp} + T_{data} + C, \qquad (1)$$

where T_{comp} is the handler processing time, T_{data} is the cost of transferring the data, and C is the additional cost from the distribution.

As a result, the total cost of *n* different handlers is:

$$T_{total} = \sum_{i=1}^{n} W_i . \qquad (2)$$

The distribution is very plausible where the gain from the usage of superior computing power is greater than the cost of the distribution. The cost, $T_{data} + C$, in equation 1, originates from transferring the message and managing the distributed computing. The other part of the equation, T_{comp}, is the computing cost of a handler. This can be decreased by using a more powerful computer. The distribution is also justified when the local computing environment does not suffice. On the other hand, there exists a game changer for the distributed environment: parallel execution.

$$T_{seq} = \sum_{i=0}^{n} T_{comp_i} \qquad (3)$$

$$T_{par} = T_{comp_0} + Max(W) \qquad (4)$$

While a sequential execution contains service and handlers' execution time, shown in equation 3, a parallel execution includes only service and the maximum execution time of the distributed handlers, demonstrated in equation 4. The speedup would be as follows:

$$Speedup = \frac{T_{seq}}{T_{par}} = \frac{\sum_{i=0}^{n} T_{comp_i}}{T_{comp_0} + Max(W)} . \qquad (5)$$

The formulas above are for a single message execution, or, in other words, for a single transaction. In the case of multiple messages, the gain can be multiplied when *pipelining* is applied. DHArch is able to improve the efficiency by offering message level pipelining for both sequential and parallel handler executions.

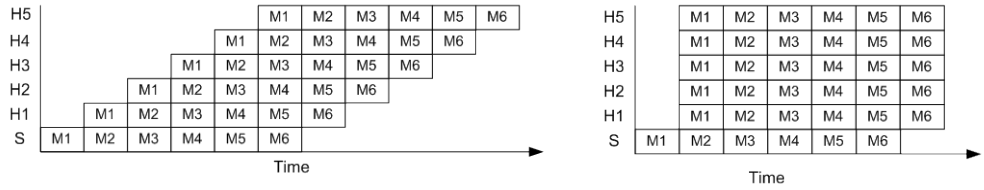


Figure 1: Web service handler pipelining, a) sequential and b) parallel

If service and distributed handlers are executing a message sequentially, as is seen in Figure 1a, the execution time of the pipelined messages will be:

$$T_{pipelineseq} = T_{comp_0} + (P - 1 + N)W \ , \qquad (6)$$

where T_{comp_0} is the service execution time, *N* is the number of messages, *P* is the number of handlers, and *W* is the distributed handler execution time.

On the other hand, when the handlers are all parallel, as is seen in Figure 1b, the execution time will be very promising.

$$T_{pipelinepar} = T_{comp_0} + NW \qquad (7)$$

3.1. Distributed Handler Architecture

DHArch has modular architecture. It employs modules so that the implementation management becomes easier and more understandable. The modules can be placed under three umbrella names: the Distributed Handler Manager (DHManager), Communication Manager (CManager), and Handler Execution Manager (HEManager), depicted in Figure 2.

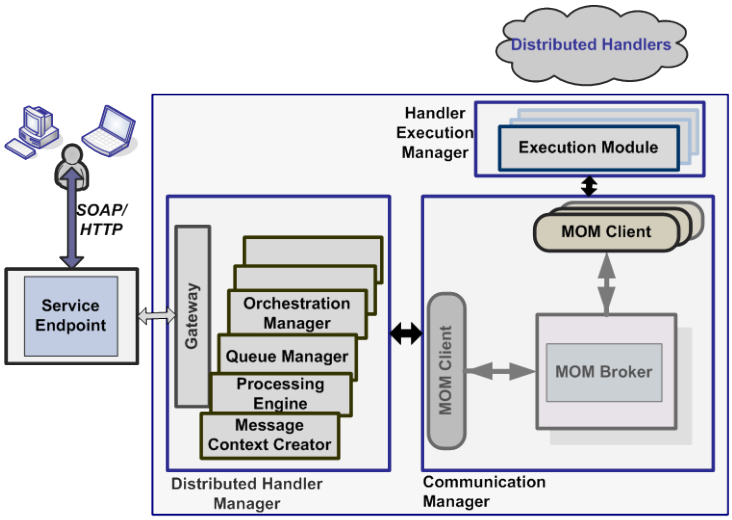


Figure 1: General Architecture of DHArch

3.1.1. Distributed Handler Manager

Distributed Handler Manager (DHManager) is an umbrella name for a group of modules that manage the message execution. It accepts messages, orchestrates the execution, and returns the output to the place from which the message was initially received. The modules in this group are called Gateway, Handler Orchestration Manager, Message Context Creator, Queue Manager, Messaging Helper, and Message Processing Engine.

Gateway is an interface between the native environment and DHArch. It is the entrance and exit point for the incoming and outgoing messages. The Handler Orchestration Manager provides the necessary orchestration capability. Since the distribution complicates the handler execution, introducing an orchestration to manage the

execution becomes necessary. The orchestration structure is investigated extensively in [11]. The Message Context Creator supports the handler execution by creating a context, the Distributed Handler Message Context (DHMContext). This context wraps the messages travelling in DHArch with supportive data. Orchestration configuration is also kept in this context so that every message has its own unique handler orchestration. Since a Web service may receive too many requests in a short time, queues are presented to regulate the message flow. The Queue Manager employs three queues: the Container Message Context Queue (CMCQueue) to store the interacting Web service container context, the Incoming Message Queue (IMQueue), and the Message Processing Queue (MPQueue) to store DHMContext. A specific format, DHArch Messaging Format (DMFormat), is created by the Messaging Helper module to facilitate the remote handler executions. It basically contains three main parts, a 128-bit unique ID, properties, and the payload. Finally, the Message Processing Engine (MPEngine) is the module that is the maestro of DHManager. It employs three threads, the Message Selector Thread (MSThread), Message Processing Thread (MPThread), and Message Receiver Thread (MRThread), to accomplish three important tasks, which are the selection of candidate messages and sending and receiving them to and from the distributed handlers.

3.1.2. Communication Manager

The Communication Manager (CManager) transports messages between the computing nodes. CManager employs NaradaBrokering, a message-oriented middleware (MOM), for the transportation [12]. NaradaBrokering provides many key messaging advantages. The first advantage is asynchronous messaging [13-16]. The requester does not idly wait for the result, but instead is notified when the output is ready. The second is regulation of the message flow [17-19]. NaradaBrokering can buffer many messages to overcome the flow in peak times. It releases these messages gradually so that the receivers can handle them. The third is a guaranteed message delivery mechanism [20-23]. We should note that reliable communication has also been investigated by the Web service community [24,25]. Finally, it scales very well because of its tree-structure broker network capability. Many brokers can link together to form a tree.

3.1.3. Handler Executing Manager

Handlers cannot perform their tasks in remote places without a supportive environment. The Handler Execution Manager (HEManager) is intended to build this necessary environment. Each distributed handler is hosted by a HEManager, which supports the execution in several ways, from negotiating with CManager for the communication to creating the necessary structures. HEManager leverages the common interfaces to standardize the handler implementation. A handler can be easily implanted in DHArch as long as it implements these interfaces. Moreover, HEManager supports some well-known handler interfaces, such as Apache Axis.

3.2. Execution

The messages arriving to DHArch are the main tasks that must be processed. Figure 3 illustrates how a message traversal happens. A message arrives within a context, specifically a Web service container context. A context consists of additional information for the execution, as well as the message itself. Since every container makes use of its own context object for the internal execution, creating a common format for the contexts requires deep knowledge about each of them. Additionally, we may end up revising the execution mechanism for each newly introduced container, and conversion between the container context objects and the DHArch-specific common format may be costly. Hence, CMCQueue is utilized to save the container contexts. At the same time, DHArch creates its unique message context, DHMContext, to perform its internal execution properly.

The message processing happens based on the guidance of the orchestration structure, which is carried out by DHMContext. The structure defines the execution sequence of the handlers. *Stage*s are introduced to support parallel execution. Many stages can be employed in an orchestration structure. Each stage should contain at least one handler, and there must be more than one handler in a stage for parallel execution. Although the stages' execution is sequential, handlers in a stage are executed concurrently.

Each created DHMContext is first stored in IMQueue. MSThread chooses a candidate from IMQueue for the execution and places it into MPQueue. The candidates are selected according to a first-come first-served scheme. It is a fair selection, because the first arriving message is chosen to be processed first [26]. However, if necessary, the selection can be done with other queuing schemes, such as priority. Utilizing two queues for DHMContext resembles the hierarchical memory structures of modern computers [27,28]. Since MPQueue is the place where the *pipelining* happens, the queues cannot be infinitely large. A mechanism similar to the TCP protocol packet rate control procedure is applied in order to ensure the best possible number of messages. It could naively be thought that it would be good idea to use a very large queue. However, we know that the access time increases when the queue length increases. More importantly, processing a tremendously crowded group of messages concurrently depletes the computing resources and causes more frequent context switches. There is a break-even point for the queue size at which the performance starts deteriorating while the queue size increases.

MPThread initiates the execution of messages as soon as the messages arrive to MPQueue. The execution is carried out by extracting necessary information from the context. With the information, DMFormat is created to transport the messages to the distributed handlers by using CManager. The handlers in one stage receive the message at the same time, although the execution may be completed at different times. MPThread waits for the completion of the distributed handler executions before starting the delivery of the message to the next stage. This procedure continues until all stages are completed. MPThread continues initiating message execution until MPQueue is empty. While MPThread tries to deplete the messages from MPQueue, MSThread stockpiles new messages on top of the queue.

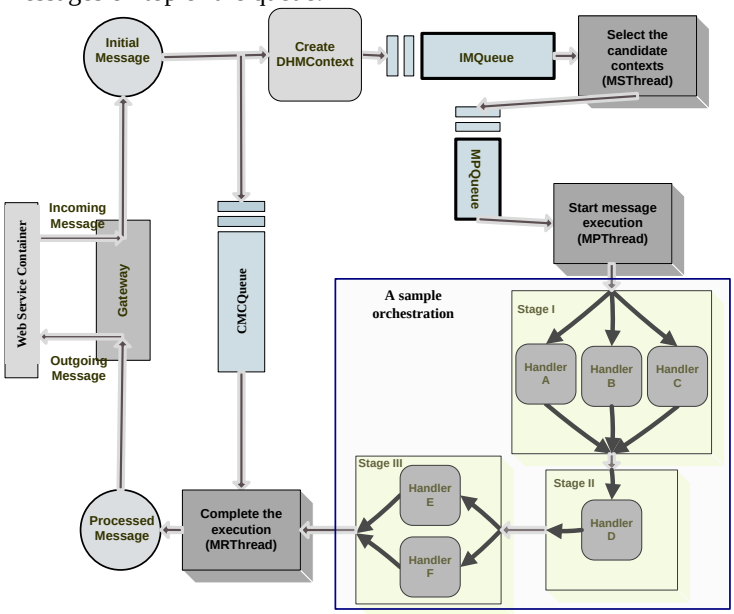


Figure 2: Message execution

DHArch threads require a clever notification mechanism. They are not allowed to run continuously. Instead, they are forced to wait if they are not needed, to avoid the otherwise inevitable wasting of system resources. If a thread continues to run with a conditional check instead of staying in its 'wait' condition, it will consume the CPU and memory resources even if it does not perform any actual task [29]. MSThread enters its wait condition when MPQueue becomes full or IMQueue becomes empty. In both situations, there is nothing for MSThread to do, so it remains in the wait condition until otherwise notified.

DHArch can utilize a wide variety of handlers, including monitoring, format converter, logging, compression, decompression, security, and reliability handlers. A handler, in general, expects the whole SOAP message as an input. On the other hand, some handlers may only process a part of the SOAP message. For example, the WS-ReliableMessaging handler processes only the *wsrm*-tagged element of the entire SOAP message. Therefore, HEManager allows the utilizing of partial execution when the size of the message is a concern. However, we need to keep in mind that partial SOAP message execution causes an overhead, originating from partitioning the SOAP message and combining the outputs later.

HEManager exploits supplementary data for the handler executions. These data are conveyed within the properties of DMFormat. Some of these properties are applicable to every handler. One of them is the *oneway* feature. It describes a situation in which a handler does not have to send any response back. When DHManager encounters a oneway handler, it applies the *fire and forget* paradigm [30] and continues its remaining tasks without waiting for a response. Additionally, the *mustPerform* property is also universal for the handlers. If a handler has a true value for the *mustPerform* parameter, it always has to complete its executions. In the event of an error, the execution has to be repeated if it does not lead to an inconstant state. Otherwise, the message execution must be completely halted and the requester must be informed.

When a handler completes its task, the output message is pushed back to HEManager. DMFormat is utilized to return the output, carried by CManager to the destination. MRThread updates the corresponding context with the executed message when it receives the output. This may not be the end of the journey; the message has to repeat these procedures for every handler in its orchestration structure. MRThread checks whether the message completes the execution for all handlers. If this is the case, the context is removed from the queue. The container context object is kept in CMCQueue until this moment, preserving the essential information to continue the message

execution in the interacting Web service container. When the container context is taken out of CMCQueue, it is modified with DHMContext, which is retrieved from MPQueue. Finally, the processed message is passed back to the point at which it was received to finalize the message execution in DHArch.

3.3. Error handling and fault tolerance

It is possible to have errors while the execution happens. If a handler stops abruptly because of a failure, the error needs to be handled so that the system can continue its execution. An error is a state that may lead to a failure. Being clear about the basis of an error is crucial to provide a solution. Laprie et al. describe two ways of dealing with failures, *fault prevention* and *fault tolerance* [31]. While the first works to prevent the occurrence of a fault, the second copes with providing the continuation of the service in the presence of the failure. Even though a complete avoidance of failure is not possible, there are tools supporting fault prevention [32]. Apparently, fault tolerance is necessary to be able to continue execution when a fault occurs. Fault tolerance requires enhancing the language to detect and handle the error. Additionally, a new semantics is essential to modify the execution on the fly.

When fault tolerance is mentioned, we need to bear in mind that forward recovery can be used as well as backward recovery. In forward recovery, an effort is made to complete the tasks by processing them several times. Backward capability requires atomicity. It is one of the most essential notions for consistency. In regard to atomicity, Hagen at al. [33] define three task types: *atomic*, *quasi-atomic*, and *nonatomic*. Atomic tasks are those that have no effect at all if they fail. For example, all read-only tasks can be thought of as atomic, because no change occurs if they fail. *Quasi-atomic* effects do not vanish naturally, although they can be eliminated via a roll-back action. *Nonatomic* tasks are those whose effects cannot be removed after they occur.

Handlers can be either stateful or stateless. A handler processes a SOAP message and applies its procedure over it. In other words, it does not keep any state for the message. This feature contributes to the utilizing of forward recovery. DHArch restarts the execution if a stateless handler fails. HEManager notifies DHManager of the error. In other words, the exception is propagated back to DHManager. DHManager starts the message execution again when it receives the exception for the stateless handler. This may be repeated several times, depending on the situation. If the execution is not successful after these efforts, the message execution is totally halted and the exception is propagated back all the way to the service requester. Handlers are not always stateless. They might be keeping states for the messages. DHArch expects atomicity from the stateful handlers. If a handler fails during its execution, it should not have any effect at all. If it is not possible to have an atomic handler or if the handler is *quasi-atomic*, it is necessary to utilize a two-phase commit. However, we prefer to employ a handler in a suitable place to commit or roll-back the effects if the handler is not atomic and stateful.

There exist cases in which the execution can continue even if an error occurs. The handler orchestration consists of a property that defines whether it is obligatory to be performed. The *mustPerform* element tells the system whether it has to be executed. If a handler contains a true value for *mustPerform*, the message execution cannot continue without achieving its execution. Otherwise, the error can be neglected and the execution continues.

4. Measurement and analysis

We perform a series of measurements illustrating the advantages of distributed and parallel handler execution in various environments. The first set of measurements is conducted to examine the performance for a single message. The second benchmark is performed by using two well-known Web service specifications, WS-Eventing and WS-Resource Framework. Finally, the scalability benchmark is used to show the efficiency of the distributed and parallel execution.

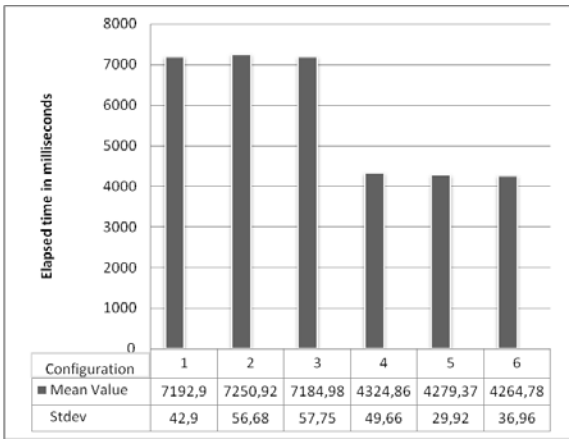
4.1. Performance measurements

DHArch is evaluated by utilizing five Web service handlers. The handlers are customized for benchmarking purposes. Two of the handlers, Handlers A and B, are CPU-bound handlers. The remaining three, Handlers C, D, and E, have been chosen from applications that gradually switch from CPU-bound to I/O-bound. The handlers are based on the tasks of decompression, decryption, writing data to file, monitoring, and logging, respectively. In order to be flexible in the configurations of the handlers, the customization of these handlers is performed so that measurements can be achieved for general purposes, without any concern about dependency. The processing time of the handlers differs. Handlers A and B require much more time to complete their tasks than the other handlers.

Out of many, six different handler combination configurations are selected for this benchmark. These six configurations are enough to reach a conclusion about the performance. The first configuration is a sequential execution of the handlers in Apache Axis. The second configuration is the sequential execution in DHArch. The sequence of the handlers is exactly the same as the first configuration. The third configuration contains two parallel and one sequential handler executions. Handler A is parallel with Handler C, and Handler B is parallel with Handler D. After the execution of these parallel handlers, handler E is executed. The fourth configuration contains the same

number of parallelisms. However, in this one, the first stage contains Handler A and Handler B, and the second stage contains Handler C and Handler D. The fifth configuration contains two stages. The handlers are executed concurrently, except Handler E, which is separated because of an imaginary dependency. Hence, it is kept for last to prevent an incorrect execution. Finally, the last configuration is created without concern about dependencies. It consists of a single stage containing five handlers. In other words, all of the handlers are parallel.

The benchmarks are conducted in three different hardware environments. To figure out the behavior of DHArch in a multi-core system, the first environment is accordingly multi-core. Nowadays, the trend is to have multi-core computers, and it is expected that more cores will be seen in single processors in the near future [34]. Hence, we give special attention to the measurements in multi-core systems. The utilized machine in this experiment has an UltraSPARC T1 processor that contains 8 cores running Solaris Operating System, 4 threads per core, with 8GB of physical memory. Although concurrent execution has many challenges [35], it activates the individual core usage in the multi-core systems; a handler may claim its own core. We can conceive of this core acquisition as if every handler has its own computing node, so that the tasks are achieved without competing for computing power. The second benchmarking environment is that the computers share a local area network (LAN). The computers in this cluster have the same hardware features. They utilize Fedora Core release 1 (Yarrow) in an Intel Xeon CPU running on 2.40 GHz and 2 GB of memory. In this environment, the handlers are distributed to the different machines. The last environment is a single computer, utilizing a Pentium 4 processor operating at 2.80 GHz with 1.5 GB of memory. It runs the Red Hat Enterprise Linux AS 4 operating system. In contrast to the previous systems, its distributed handlers need to share a single computing resource.



Configuration	1	2	3	4	5	6
Mean Value	7192,9	7250,92	7184,98	4324,86	4279,37	4264,78
Stdev	42,9	56,68	57,75	49,66	29,92	36,96

Figure 3: The execution of a Web service containing the five handlers with six handler configurations in the multi-core system

The measurement shown in Figure 4 depicts the results from the multi-core system. The values show the round-trip time of a service request. Clients record the time of the request initiations and calculate the elapsed time when they receive the responses. Hence, the measurements contain transportation, service, and handlers' execution times. Every observation is repeated 100 times. It is clearly seen that the best results are observed when all handlers are able to run concurrently. The difference between configurations 1 and 2 is the overhead originating from the distribution of the five handlers. The first configuration utilizes Apache Axis in-memory handler deployment. In the second configuration, DHArch increases the execution time slightly because of the distribution of the handlers to the individual cores. The gain is tiny in configuration 3 because of the processing times of Handlers C and D. As a result, this configuration slightly provides enough gain to overcome the overhead. Sometimes, a gain may not even compensate the overhead. On the other hand, a configuration may provide a fascinating performance with the execution of the handlers in a parallel manner, as in configurations 4, 5, and 6.

Figure 5 illustrates results from the executions of the handlers in the cluster that communicate via the LAN. The execution times get smaller due to faster computers. However, this does not change the behavior of the handler configurations. They follow the same patterns of the multi-core system. The sequential execution of DHArch is slower than those of the other configurations.

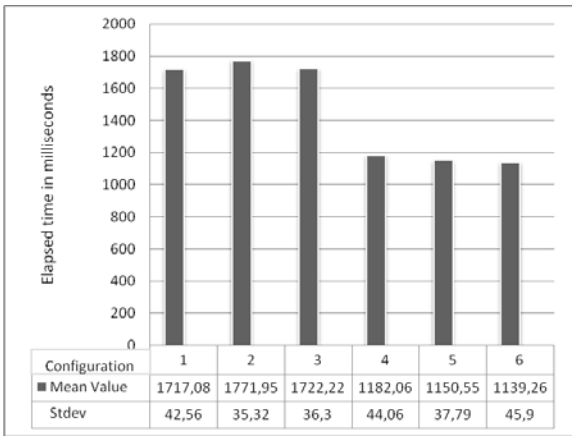


Figure 4: The execution of a Web service containing the five handlers with six handler configurations in the cluster utilizing a local area network

The results shown in Figure 6 are from the single processor system. In contrast to previous measurements, a single processor system yields a different pattern. Thread scheduling becomes an issue. Since two handlers are heavily CPU-bound, their individual execution times increase when they are executed concurrently. Moreover, NaradaBrokering and Apache Axis in an Apache Tomcat container use the same processor. This situation worsens the context switch issue.

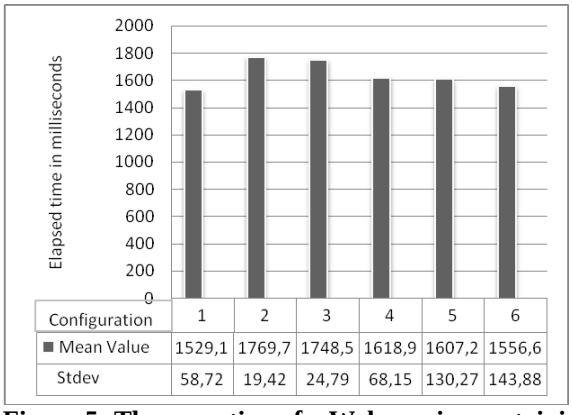


Figure 5: The execution of a Web service containing the five handlers with six handler configurations in the single processor system

The management of the distributed handler execution and the transportation of tasks increase the execution time. However, there are ways of compensating for the overhead and even achieving a promising overall performance. The first is to utilize more powerful computing resources. The second is to establish concurrent handler execution in the distributed environment. The benchmark provides enough evidence that the distribution and parallel execution of appropriate handlers can provide very plausible results. When the single processor system results are investigated, the necessity of the distribution is clearly seen. The processor is saturated because of too many tasks to complete. The same anomaly is not witnessed in the multi-core system or the cluster utilizing a LAN, because they offer more computing power.

4.2. Deploying Web Service Resource Framework and Web Service Eventing

In the previous benchmark, the customized handlers were utilized to see the output patterns in various combinations of parallel handlers. For this benchmark, we want to show the deployment of two well-known Web service specifications to provide a concrete example. We have found several groups providing WS-specification implementations. Among them, two specs were fitting to our purpose: WS-Resource Framework [36] and WS-Eventing [37]. Web Service Resource Framework (WSRF) establishes the necessary standards to manipulate *states*. It basically offers capabilities to insert, update, and discover stateful resources in a standard and interoperable way. For the benchmark, Apache implementation of WSRF is utilized [38]. On the other hand, Web Service Eventing (WS-Eventing) defines a protocol to standardize notification efforts. A Web service may benefit from receiving notification when an event occurs. Instead of checking an event occurrence repeatedly, an entity can be notified by an *event source* when an event happens. In this paradigm, a service, called a *subscriber*, needs to register itself to a

certain interest with another service. An implementation of WS-Eventing from the Pervasive Technology Lab at Indiana University is utilized for this benchmark [39].

First of all, the specifications are initialized. Sink registers itself to the topic */sensor/california* and a sensor stateful resource stores the initial information. Messages combining WS-Eventing and WSRF are created in order to run the handlers in a parallel manner. The message notifies of an important activity and updates information for a sensor stateful resource. When it is received, the WS-Eventing source handler looks for the subscription manager service to find the interested subscribers, the sink. It then delivers the event to them. While notification is happening, the WSRF handler also updates the values of the states, which are kept in storage, and forwards the information with the additional data previously stored.

A computer cluster containing 8 machines, which have the same features, is employed. They share a LAN to communicate with each other and utilize Fedora Core release 1 (Yarrow) in an Intel Xeon CPU running on 2.40 GHz and 2 GB of memory. We first gather the results for Apache Axis by running WS-Eventing and WSRF sequentially. The handlers are deployed into the request path. We individually measure their execution times, as shown in Table 1. Each request is observed 100 times. We also perform the same sequential handler execution in DHArch. Because of the overhead originating from the distribution of the handlers, the time to process a single message increases.

Table 1: WSRF and WS-Eventing handler execution

		WSRF	WS-Eventing	Total service
Axis sequential	Execution time (ms)	65.24	51.38	161.23
	Standard deviation	7.39	5.42	9.66
DHArch sequential	Execution time (ms)	70.25	54.68	171.64
	Standard deviation	4.45	3.93	10.08
DHArch parallel	Execution time (ms)	69.49	54.45	115.15
	Standard deviation	5.53	3.42	12.15

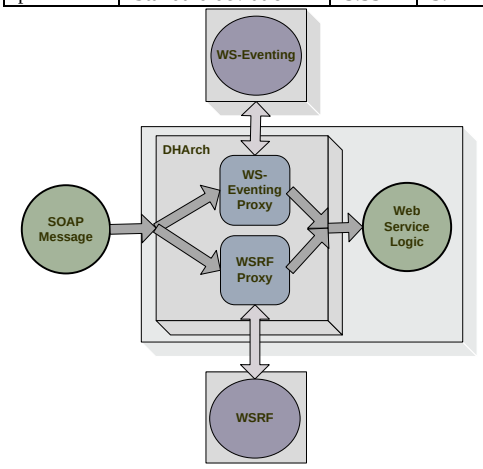


Figure 6: Parallel execution of WSRF and WS-Eventing

When we introduce the parallelism, as portrayed in Figure 7, we see a significant improvement in the service performance. The concurrency reduces the execution cost of a single request by one-fourth. WSRF is the main player in determining the processing time of the handlers joined to the parallel execution, because its processing time is the highest. Table 1 shows the execution times and standard deviations of the handler executions.

4.3. Scalability

Having obtained results on distributed and parallel executions for a single Web service interaction, we now investigate the throughput compared to a conventional handler structure. We also examine the effect of the request rate over the processing time. A Web service is basically a paradigm in which clients make requests to execute a task in a remote application. This structure may lead to a situation in which many clients make many requests in a short amount of time. For instance, an online shopping center that utilizes Web service technology may receive hundreds of transactions. Consequently, a Web service may have a very high request rate; the system architecture must be efficient and effective to answer the increasing number of requests.

Eight clustered multi-core computers communicating via a LAN are utilized for benchmarking purposes. Every computer has two Quad-core Intel Xeon processors running at 2.33 GHz with 8 GB of memory, operating Red Hat Enterprise Linux ES release 4. Three handlers are employed: Logger, Monitor, and Format Converter. Logger stores the incoming messages in a file. Monitor keeps the information for the services, such as the incoming

message rate, the message size, information about the clients, number of clients that are connected, and so on. The last handler, Format Converter, converts incoming message formats, which may be coming from different sources, to a uniform format, a format that the service expects. The messages are sent at the same rate, starting from one message per second. The number of messages is continually increased by ten in every step, up to the level that the service can support. We collect the execution times for a single message while the number of messages per second is increasing. DHArch results are gathered for parallel as well as sequential executions, portrayed in Figures 8 and 9, while Apache Axis is employed only for sequential execution. The gathered results are shown in Figure 10.

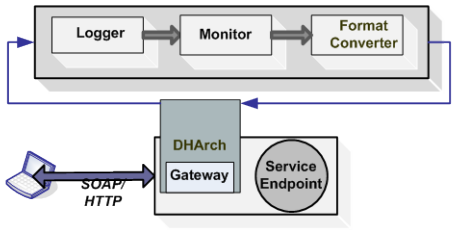


Figure 8: DHArch sequential handler deployment for the scalability measurement

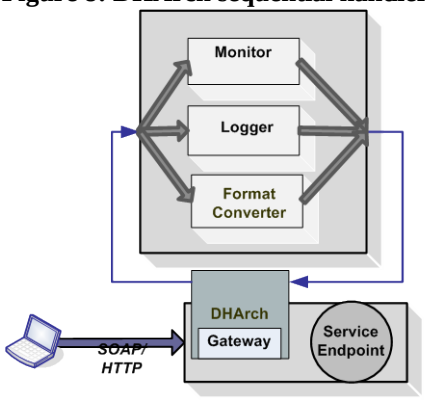


Figure 9: DHArch parallel handler deployment for the scalability measurement

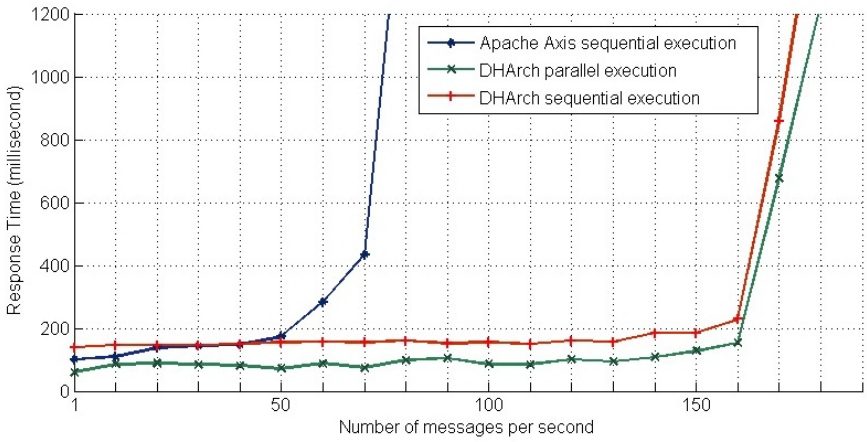


Figure 70: Response time of a single request for an increasing number of messages per second in clustered multi-core machines communicating via LAN

For the Apache Axis deployment, the execution time is continually increasing because every additional message shares the computing resources. It catches and surpasses the DHArch sequential execution quickly. However, it does not cause any abrupt changes until the resources are completely consumed. When the resources become unable to meet the demands, the execution time skyrockets. The problem is that there are too many pipelined messages running on the available computing resources. On the other hand, the processing time in DHArch stays stable longer, because DHArch can benefit from additional computing power. The message rate does not change the response time until it reaches 160 messages per second. Furthermore, messages that the system cannot support are forced to wait in the queue to optimize message execution.

Between the sequential and parallel execution in DHArch, there is always a difference. For each message execution, there exists a gain coming from the parallel execution. The difference is stable and persistent. When this difference is considered for a series of message executions, the advantage of parallel execution is clearly seen. The parallelism brings a more than 30% gain for each message execution, even at the rate of 160 messages per second. Of course, this gain depends on the distributed handlers and their combination for parallel execution.

5. Conclusion

Web services exploit additive functionalities, Web service handlers, to improve capabilities such as security, reliability, and logging. In many cases, the functionalities are very essential for the services. This requirement forces the Web service execution environment to evolve into the direction of efficiency and scalability. The design of the executing environment is very critical for success. Therefore, we have built Distributed Handler Architecture (DHArch) to investigate the promising environment and derived important conclusions from this conclusive research.

Distributed computing provides very efficient and scalable leverage for Web service executions. Since a Web service can contain many handlers in addition to the service endpoint, together they may saturate a single computing entity, as shown in Figure 6. This gets worse when many clients simultaneously make requests from the service point running on a single machine. Therefore, it is necessary to provide additional computing power. Web service handlers need to utilize not only a single computer, but also multi-core and clustered computers. Accepting a Web service containing endpoint logic and handlers as a single entity prevents the utilization of additional computing power. Instead, handlers can be separated from the endpoint logic so that they can run on different cores or computers. Our benchmark results show the advantage of handler execution in the distributed environment. Clustered computers on a LAN offer a very efficient environment for the handlers, since the network latency is so minimal that it can be ignored. On the other hand, especially for the CPU-bound handlers, utilizing a separate core results in very appealing outputs, thanks to the existence of distributed computing without network latency.

Orchestration is a significant feature for collaboration among the distributed applications. A promising result cannot be expected without a decent orchestration mechanism, which must provide several main features for the handler distribution. First of all, it needs to offer a very efficient and effective orchestrating engine. We choose an approach to separate the description from the execution, which reduces the complexity of the engine while providing a powerful expressiveness for the handler orchestration. Secondly, the orchestration mechanism needs to build a dynamic and adaptive handler execution structure. The modification of a handler execution sequence should be permissible unless the correctness of the execution is not intact. Finally, an individual handler sequence for a specific message should be allowed. In other words, every message can execute its own handler orchestration flow.

Web service handler architecture needs to have the ability to exploit parallel execution. Even though not all handlers are suitable for execution in a parallel manner because of dependencies and the execution order that must be followed, there are many handlers available for parallel execution without harming the correctness of the execution. However, the configuration of handler executions must be selected carefully. As we can see from the benchmarks, only some handler configurations out of many provide promising outputs.

Moreover, message level parallelism, in addition to the parallel handler execution, should benefit. Instead of waiting for the completion of a single message execution, many messages can be processed simultaneously, which is called *message pipelining*. However, only an optimum number of requests should be executed concurrently. The remaining requests should be kept in a queue instead of letting every message arriving to the system start its execution right away. This flow regulation prevents performance degradation, as shown in Figure 10. The benchmark results prove that the pipelined messages in the distributed environment scale much better than the conventional handler structures.

References

1. Web Service Architecture, http://www.w3.org/TR/ws-arch.
2. Simple Object Access Protocol (SOAP), http://www.w3.org/TR/soap12-part1.
3. Web Service Description Language (WSDL), http://www.w3.org/TR/wsdl.
4. Universal Description Discovery and Integration (UDDI), http://www.uddi.org.
5. Hoare, C.A.R., *The Emperor's Old Clothes*. 1981, ACM Press, New York, NY, USA. pp. 75-83.
6. Apache Axis, http://ws.apache.org/axis.
7. Perera, S., C. Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, and G. Daniels, *Axis2, Middleware for Next Generation Web Services.* IEEE International Conference on Web Services (ICWS'06), 2006, Chicago, USA.
8. Shirasuna, S. et al., *Performance Comparison of Security Mechanisms for Grid Services.* 5th IEEE/ACM International Workshop on Grid Computing, pp. 360- 364, Nov. 2004.
9. Slominski, A. et al., *Asynchronous Peer-to-Peer Web Services and Firewalls*. 7th International Workshop on Java for Parallel and Distributed Programming, 2005.

10. Fox, Geoffrey C., *Lessons from Massively Parallel Applications on Message Passing Computers*. Proc. 37th IEEE International Computer Conference, 1992.
11. Yildiz, B., G. Fox, and S. Pallickara, *An Orchestration for Distributed Web Service Handlers.* The Third International Conference on Internet and Web Applications and Services, 2008, Athens, Greece.
12. Fox, G., S. Pallickara, and S. Parastatidis, *Toward Flexible Messaging for SOAP-Based Services.* The 2004 ACM/IEEE Conference on Supercomputing, 2004.
13. Aoyama, M. and A. Mori, *A Unified Design Method of Asynchronous Service-Oriented Architecture Based on the Models and Patterns of Asynchronous Message Exchanges*, pp. 537-544, The IEEE International Conference on Web Services, 2008.
14. Hu, J., Z. Zeng, G. Zhao, C. Long, and F.E Luo, *s-AMM: A Service-oriented Asynchronous Messaging Middleware*, Vol. 2, pp. 312-316, The Second International Symposium on Electronic Commerce and Security, 2009
15. Langendoen, K., R. Bhoedjang, and H. Bal, *Models for Asynchronous Message Handling*, IEEE Concurrency, Vol. 5, No. 2, pp. 28-38, Apr.-June 1997.
16. Nakao, T. and S. Yokoyama, *An Asynchronous Messaging Platform for Development of Context-aware Services*, pp. 451-460, The 8th International Symposium on Autonomous Decentralized Systems (ISADS'07), 2007.
17. Qiu, D. and N. Shroff, *Queueing Properties of Feedback Flow Control Systems*, Vol. 13, pp. 57-68, The IEEE/ACM Transactions on Networking, 2004.
18. Wang, W., M. Palaniswami, and S.H. Low, *Application-oriented Flow Control: Fundamentals, Algorithms and Fairness*, Vol. 14, pp. 1282-1291, The IEEE/ACM Transactions on Networking, 2006.
19. Qiu, D. and N.B. Shroff, *A New Predictive Flow Control Scheme for Efficient Network Utilization and QoS*, pp. 143-153, ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, New York, NY, 2001.
20. Fox, G., *A Scheme for Reliable Delivery of Events in Distributed Middleware Systems*, pp. 328-329, The First International Conference on Autonomic Computing (Icac'04), 2004, Washington, DC.
21. Pallickara, S. et al., *On the Costs for Reliable Messaging in Web/Grid Service Environments.* IEEE International Conference on e-Science & Grid Computing, pp. 344-351, 2005, Melbourne, Australia.
22. Tai, S., T.A. Mikalsen, and I. Rouvellou, *Using Message-oriented Middleware for Reliable Web Services Messaging.* Lecture notes in computer science, ISSN 0302-9743, 2003.
23. Halle, S. and R. Villemaire, *Flexible and Reliable Messaging using Runtime Monitoring*, The 13th Enterprise Distributed Object Computing Conference Workshops (EDOCW 2009), pp. 116-125, 2009.
24. Web Service Reliable Messaging (WS-ReliableMessaging), ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200502.pdf.
25. Web Services Reliability (WS-Reliability), http://www.oracle.com/technology/tech/webservices/htdocs/spec/WS-ReliabilityV1.0.pdf.
26. Bajpai, R., K.K. Dhara, and V. Krishnaswamy, *QPID: A Distributed Priority Queue with Item Locality*. The International Symposium on Parallel and Distributed Processing with Applications, pp. 215-223, 2008.
27. Teng, W.G., C.Y. Chang, and M.S. Chen, *Integrating Web Caching and Web Prefetching in Client-side Proxies*. The IEEE Transactions on Parallel and Distributed Systems, Vol. 16, No. 5, pp. 444- 455, 2005.
28. Luican, I.I., H. Zhu, F. Balasa, *Formal Model of Data Reuse Analysis for Hierarchical Memory Organizations*, The IEEE/ACM International Conference on Computer-Aided Design (ICCAD '06), pp. 595-600, 2006.
29. Silberschatz, A., P.B. Galvin, and G. Gagne, *Operating System Concepts*. 2002, Addison-Wesley. Reading, Massachusetts.
30. Voelter, M., M. Kircher, and U. Zdun, *Patterns for Asynchronous Invocations in Distributed Object Frameworks*, The Eighth European Conference on Pattern Languages of Programs (EuroPLoP 2003), 2003.
31. Laprie, J.C.C., A. Avizienis, and H. Kopetz, *Dependability: Basic Concepts and Terminology*. 1992, Springer-Verlag New York, Inc. Secaucus, NJ, USA.
32. Leymann, F. and W. Altenhuber, *Managing Business Processes as an Information Resource*. IBM System Journals, Vol. 33, Issue 2, pp. 326-348, 1994.
33. Hagen, C. and G. Alonso, *Exception Handling in Workflow Management Systems*. IEEE Transaction on Software Engineering, Vol. 26, Issue 10, pp. 943-958, 2000.
34. Johnson, C. and J. Welser, *Future Processors: Flexible and Modular.* The 3rd IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (*CODES+ISSS '05* 2005), 2005.
35. Majumdar, S., D.L. Eager, and R.B. Bunt, *Scheduling in Multiprogrammed Parallel Systems.* SIGMETRICS Perform. Eval., Rev. 16, 1, pp. 104-113, 1988.
36. Web Service Resource Framework (WS-Resource Framework), http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf.
37. Web Service Eventing (WS-Eventing), http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf.
38. Apache WSRF, an implementation of WS-Resource Framework, http://ws.apache.org/wsrf.
39. FINS, an Implementation of WS-Eventing, http://www.naradabrokering.org/FINS-Docs.