DISTRIBUTED HANDLER ARCHITECTURE

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

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

Abstract

Over the last couple of decades, distributed systems have been demonstrated an architectural evolvement based on models including client/server, multi-tier, distributed objects, messaging and peer-to-peer. One recent evolutionary step is Service Oriented Architecture (SOA), whose goal is to achieve loose-coupling among the interacting software applications for scalability and interoperability. The SOA model is engendered in Web services, which provide software platforms to build applications as services and to create seamless and loosely-coupled interactions. Web services utilize supportive functionalities such as security, reliability, monitoring, logging and so forth. These functionalities are typically provisioned as handlers, which incrementally add new capabilities to the services. Even though handlers are very important to the services, the way of utilization is very crucial to attain potential benefits. Every attempt to support a service with an additive functionality may increase the chance of having an overwhelmingly crowded chain. Moreover, a handler may become a bottleneck because of having a comparably higher processing time. We present Distributed Handler Architecture to provide an efficient, scalable and modular architecture to address these issues.

Keywords: Service Oriented Architecture, Web Service, Parallel Computing, Pipelining, Handler, Container

1. Introduction

One recent evolutionary step in computing environment is Service Oriented Architecture (SOA) whose goal is to achieve loose coupling, scalability and interoperability. SOA manifests itself perfectly in Web services, supplying platforms to build applications as services. Web service framework offers standard ways to interoperate among software applications, running on a variety of platforms [1]. It provides seamless and loosely coupled communications; applications can communicate with each other without giving much effort even though they might be utilizing different languages and platforms.

Web service provides a common ground to offer interoperability. Many standards have been introduced and many of them are on the way. The key features of the Web services, which are described by World Wide Web Consortium (W3C), have been introduced as Web service specifications. Simple Object Access Protocol (SOAP)[2], Web Service Description Language (WSDL) [3], and Universal Description Discovery and Integration (UDDI) [4] are de-facto standards.

One of the most crucial aspects of Web service framework is the utilization of the XML messaging. SOAP is an XML based data exchange format, which is employed by Web services. Consequently, Web service framework heavily depends on SOAP processing. As a result, several Web service containers, the middleware in Figure 1, has been introduced to take pressure off the applications. Their main goal is to hide the details of the SOAP processing from the users. The most popular containers are Apache Axis[5], Microsoft Web Service Enhancement[6] and IBM Websphere[7] .

The container architecture employs two main SOAP processing components, Web service endpoint logic and handler. Handler is also called as filter. Web service endpoint logic, which is a standalone application, carries out main task. On the other hand, a handler is a supportive application. It contributes to a service with additional capabilities such as reliability, security and logging.

Despite the fact that handlers preferably deal with header, they also have the ability to modify SOAP body. In addition to de-facto standards, many WS-specifications have been introduced so far. They are the efforts where the community sets the standards to have more interoperable systems[8]. Some of them are very good candidates to be handlers, especially, those dealing with the headers.

Web services are able to employ a set of handlers to acquire many capabilities in a single execution. For instance, a service may need to be reliable as well as secure at the same time. Handler chains are introduced for this purpose. Container engines let a message travel through handlers in a chain.

Apparently, handler is a crucial aspect of Web Service Architecture because of the key importance in the execution path. However, the way of utilizing 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 in the execution pipeline. For instance, reliability adds significant amount of processing time. Similarly, security may necessitate powerful machines to conclude its task in a reasonable time. Any additional handler may make the response time of a service worse.

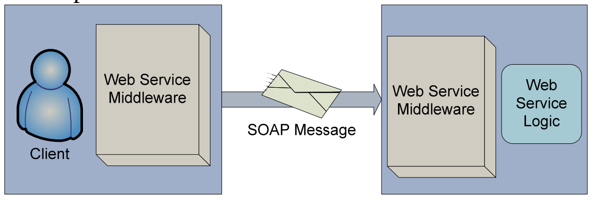

Figure 1: A Simple Web Service Interaction

Nevertheless, 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 in their execution paths. Accordingly, we may wind up with an overwhelmingly crowded pipeline of the handlers. This circumstance will make the services slower. This situation is named as a Web service becomes *fat*; while the service is acquiring new capabilities, the response time becomes longer and the management of the service becomes harder. 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, a handler becomes *bottleneck*. This condition mounts the request messages waiting to be served in every second. The clients start waiting longer and longer.

Let's think about a highway, which has three lanes. And it is rush hour. Everybody is driving to reach home and get relaxed as soon as possible. However, at some point, the road becomes narrower. Since it is peak time; the road capacity is not sufficient to serve the arriving cars. In every passing minute, the number of cars grows. The people start becoming distressed because they do not want to waste their time in the highway by just waiting. The first solution is to expand the narrow part of the road. Adding one or two lanes to the narrow part will suffice. The second solution is to detour a portion of the traffic to a parallel road. We can utilize both approaches in the handler architecture. Replacing the narrow road resembles introducing new enhanced computing environments. Using the parallel road looks like offering concurrent execution for the handlers.

We have additional resources out there. Networks are becoming faster. Machines are becoming more powerful and their speed is constantly improving. Hence, these improvements can contribute to remove insufficient parts. Bottlenecks can be eradicated by delivering some of the handlers to the powerful computers. The distribution reduces the burden over a single computer.

Application parallelism is not new idea; it has been utilized for decades. Hence, handlers can be executed concurrently. However, handler parallelism is not able to be utilized in the conventional Web Service Architecture. The parallelism boosts the performance and provides very effective and powerful solution.

Recently, an enhancement in processor technology becomes popular. Multi-core processors are started being widely utilized; even personal computers leverages cores offering opportunity for parallel executions. This opportunity contributes to the parallel handler execution even without introducing any network latency.

Distribution of the applications is very crucial to improve performance and scalability. However, there are requirements to be able to benefit from it. The decision of a handler distribution is influential over the system performance. Moreover, the selection of the handlers running concurrently is very important. The conditions and requirements of the distribution of a handler are necessarily needed to be investigated extensively. Handler structure demands efficient handler orchestration. The handlers have to be orchestrated in a way that Web Service benefits most. The orchestration is especially essential when the handlers are distributed. It becomes inevitable, when the concurrency is launched for the handler executions.

Reusability is one of the key features for an application. Instead of deploying the same handler many times, we may make use of a handler repeatedly. There are many stateless handlers. They process a SOAP message and return the results without keeping any information for requester. For instance, compression and decompression are stateless functionalities. Hence, they are very suitable to be used by the services and/or clients many times without complications. Even stateful handlers may become appropriate to be utilized repeatedly in certain conditions.

Handlers offer new capability without increasing the complexity. Simplicity is a very crucial feature of applications. In Web Service structure, simplicity originates from very well known notion, *divide and conquer.* The whole task is divided between handlers and the service endpoint. Instead of having a large, hardly manageable application, clearly separable smaller tasks are more plausible. Charles Antony Richard Hoare states this very essential feature to design excellent software in his *The Emperor's Old Clothes* [9]. He says that "*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.*" Simplicity contributes to constructing modular and flexible applications. However, it is a challenging effort to build a perfectly flexible and modular system.

Consequently, handler architecture needs to be investigated to provide efficient, scalable and flexible Web services. Since a SOAP task, which is either related with the body or header, may be costly, we need additional resources and structures. We can improve the performance, make the system scalable and provide improved architectures.

2. Handler Structures

There are several conventional Web service handler structures which provide an environment to add new functionalities to Web service end-point. The first structure, worth to mention, is JAX-RPC. It offers necessary tools to deploy handlers, shown in Figure 2. Handlers can construct handler chains in both client and server sides. The executions are sequential and deployment is static; the execution path cannot be modified after being deployed.

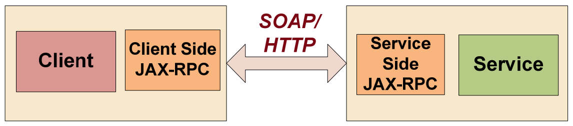

Figure 2 : JAX-RPC architecture

Apache Axis is currently the most dominant container in the Web service community and has a plethora of applications developed around this container. There are two main versions, Apache Axis 1.x and Apache Axis 2.

Apache Axis 1.x facilitates the incremental addition of capabilities to Web service endpoint by leveraging handlers. Handlers can be either request or response path. At one point, a handler sends request as well as receives response. This handler is called pivot handler. It processes requests and passes them to the endpoint. When the endpoint finishes its tasks, the responses are sent back to the pivot handler. There exist two types of handlers. The first type contains singleton handlers, which do not require a peer. They can be deployed to either client or server side. On the other hand, there are handlers that necessitate peers in the client and the service sides. For instance, an encryption handler which encrypts messages coming from a client requires an inverse handler at the service side which performs the appropriate decryption. Client side handler peers are processed in the reverse order of the service side handler peers. For example, if a client processes handlers in the order of h1, h2 and h3, their counterparts in the service side are executed in the order of h3, h2 and h1.

Apache Axis 2 has more extensible and modular architecture. The core modules are separable from the remaining modules so that the new modules can be added on the top of the core modules [10]. To handle information and keep the states, Apache Axis 2 defines an Information Module. Information module has a hierarchical structure that helps to manage the object lifecycles. Apache Axis 2 basically views every transaction as a single SOAP processing. To implement a complex SOAP messaging, containing several messages, a top layered framework is necessary. Apache Axis 2 framework contains two pipes: IN and OUT. They may be combined to exchange messages. User application can create a SOAP request by using a client API. Before handing the message over transport sender, new capabilities can be added with the handlers. They provide extensibility to the SOAP processing model. They can intercept messages in either IN or OUT pipe.

Additionally, Apache Axis 2 introduces an upper level abstraction on to top of handler layer: 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. They are basically intended to implement Web service specification in a modular manner such as WS-Addressing [11] and WS-Reliable Messaging[12].

There are stages to arrange the order of the modules. These stages are called as phases. Phases and flows together manage the processing flow for a specific message. Apache Axis 2 contains predefined special handlers such as Dispatchers, Transport receiver and Transport sender. Similarly, several predefined special phases are also introduced: Transport, Pre-Dispatch, Dispatch, User defined and Message Processing phases in IN pipe and Message Initialization, User and Transport phases in OUT pipe. However, this mechanism is not fixed; it is extensible and user customized phases and handlers are allowed to be attached.

Similar to Apache Axis, Web Service Enhancement (WSE) from Microsoft supports Web services by offering an environment for the supportive capabilities, which are called *filters*. The execution structure of the filters is very similar to that in Apache Axis. Both Output and Input filters are capable of processing SOAP header and body. The real target is the header, though. WSE has already several build-in filters. However, customizable filters can be added. Filters can create a chain. In this chain, the intermediary information is passed between the filters by using a context. The context provides an environment to share the properties and variables.

Finally, DEN/XSUL provides architecture which offers an environment for the handler execution. XSUL is a modular Java Library to construct Web and Grid services [13, 14]. It has been developed by Extreme Lab at Indiana University and 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 [15]. It targets directly to the Web service security processing steps 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 Grids. The whole scenario is depicted in Figure 3. XSUL2, the latest version of XSUL, allows a request goes through a chain of handlers until it reaches the destination. DEN by utilizing XSUL2 is able to separate the handlers from Web Service endpoint and distribute them as individual service nodes within a chain.

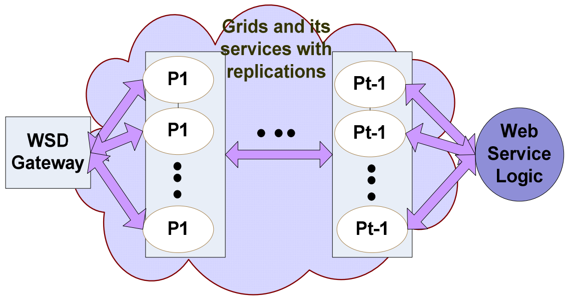

Figure 3 : DEN Web service execution

DEN/XSUL is able to utilize asynchronous messaging by using WS-Dispatcher. The dispatcher allows internal services to be exposed to Internet. With the support of asynchronous messaging, a WS-Security implementation is divided into sub-atomic tasks and deployed as services. Some tasks are executed in a parallel manner to gain performance and to remove the bottleneck.

While DEN/XSUL touches the parallel execution by distributing the subtasks, we cannot witness that the parallelism is benefited enough. On the other hand, the containers, Apache Axis and WSE, use pipelining to run a handler chain although they do not have any parallelism in the handler executions.

3. Distributed Handler Architecture

In order to gain full benefit from Web services for the handler execution, we introduce Distributed Handler Architecture (DHArch). It is a framework that provides functionalities to process handlers concurrently as well as sequentially in a distributed environment. The goal is to remove the boundaries that keep the handlers in a single memory space and to contribute to the modularity, reusability, interoperability, scalability and responsiveness of the system.

Figure 4 depicts the overall picture of DHArch. It has modular architecture. Instead of having a very big chunk of hardly manageable implementation, DHArch employs modules so that the implementation management became easier and more understandable. DHArch modules can be placed under three umbrella-names: Distributed Handler Manager (DHManager), Communication Manager (CManager) and Handler Execution Manager (HEManager).

3.1. Distributed Handler Manager

Distributed Handler Manager (DHManager) is a group of modules that manages message execution. It is mainly the hearth of system; it accepts messages, orchestrates the execution and returns the output to the place where the message initially has been received. It contains sub-modules: Gateway, Handler Orchestration Manager, Message Context Creator, Messaging Helper, Queue Manager and Message Processing Engine.

Gateway is an interface between the native environment and DHArch. It is entrance and exit point for the incoming and outgoing messages. DHArch has a native environment independent architecture. It autonomously performs the given tasks. However, Gateway module is an exception. Since it connects DHArch to the underlying environments, it utilizes the libraries and tools of the native environments.

DHArch is a system enabling the execution of the handlers in a distributed fashion. This environment brings many advantages such as utilization of additional resources and concurrency. On the other hand, the distribution complicates the handler execution. Handlers became unaware of each other when they are scattered around. Therefore, introducing an orchestration to manage the execution becomes necessary. Handler Orchestration Manager provides the necessary orchestration capability. The orchestration structure is investigated extensively [16].

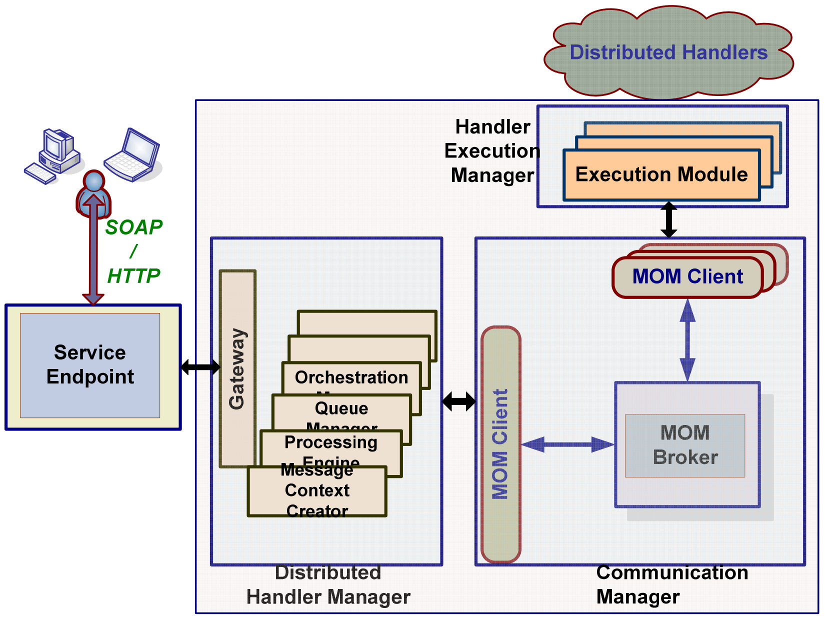

Figure 4: General Architecture of DHArch

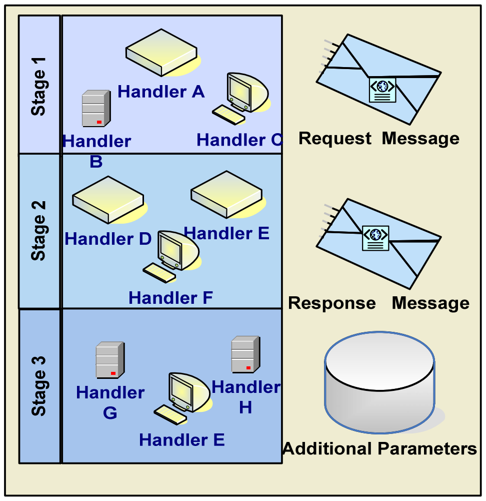

Figure 5: Distributed Handler Message Context

Message Context Creator supports the handler execution by creating a context, Distributed Handler Message Context (DHMContext), shown in Figure 5. This context wraps the messages travelling in DHArch. Additionally, it conveys supplementary information for the execution such as current stage number, number of handlers, number of stages, start and end times of handler execution and so on. Orchestration configuration is also kept in the context. Every message can have its unique handler orchestration.

A Web Service may receive too many requests in a short duration. Hence, queues are introduced to regulate the message flow. It is similar to having a waiting room in a doctor office. When patient arrives, s/he is asked to fill the necessary information and to be seated in the room until the doctor becomes available. Similarly, DHArch registers the necessary information and makes the message wait to be called. Queue Manager employs three queues. The first queue, Container Message Context Queue (CMCQueue), stores the interacting Web service container contexts. For example, *MessageContext* is the context object of Apache Axis container to be stored. In contrast to CMCQueue, Incoming Message Queue (IMQueue) and Message Processing Queue (MPQueue) store DHMContext, which is created by Message Context Creator module. IMQueue stores a DHMContext object for every arriving message. On the other hand, MPQueue only keeps the message contexts which are being executed. Therefore, the number of the messages in the queue is limited for the optimization purpose.

Messaging is a very significant capability to decouple the computing nodes. Sending and receiving the tasks between the interacting nodes via messaging contributes to the interoperability. Although the messages can be sent in different formats, a specific format, DHArch Messaging Format (DMFormat), is created by Messaging Helper module to facilitate the remote handler executions. It basically contains three main parts, unique ID, properties and payload. Each DMFormat contains a 128-bit unique ID, created by a UUID generator. The uniqueness keeps the message execution intact so that the message execution cannot possibly interfere with another. Properties convey the required information for the computing nodes: Handler Execution Manager (HEManager) and Distributed Handler

Manager (DHManager). Payload contains the original message. There isn't a restriction for the payload format; any kind of message format can be embedded to the payload.

Message Processing Engine (MPEngine) is the maestro of DHManager. It employs three threads to accomplish three important tasks to orchestrate the execution: selecting candidate messages, sending messages to the distributed handlers and receiving the responses returning from the distributed handler. Message Selector Thread (MSThread) selects a DHMContext instance of candidate message from IMQueue and puts it into MPQueue. On the other hand, Message Processing Thread (MPThread) takes the context from MPQueue and extracts the required information to initiate the transportation of the payload. The third thread, Message Receiver Thread (MRThread), become active when an executed payload is received. It checks the message ID to match the corresponding context in the MPQueue and updates the context with the response. If every handler has completed its task, MRThread removes the context from MPQueue, combines it with the native context of CMCQueue and returns the output to Gateway module.

3.2. Communication Manager

Communication Manager (CManager) transports the messages between the computing nodes. A Message Oriented Middleware (MOM) is employed for the transportation. We use NaradaBrokering for this purpose [17]. It provides many key advantages for the messaging. The first advantage is asynchronous messaging [18-21]. While requester is asking a service, the provider can be in the situation of performing another job. This is also called as non-blocking IO [22]. The requester does not wait for the result; it is notified when the output is ready. This eliminates the idle waiting. The second advantage is to regulate the message flow. Flow control has been widely investigated[23-26]. NaradaBrokering can buffer so many messages to overcome the flow in peak times. It releases these messages gradually so that the receivers are able to handle the messages. The third advantage is to have a guaranteed message delivery mechanism[27]. There exist many researches in this area[28, 29]. Web service community has recently introduced specifications for the reliable communication too [30, 33]. NaradaBrokering provides a robust delivery mechanism by storing messages in a database so that the peers can get them later even if a failure occurs. Moreover, it scales very well because of tree structure broker network capability. Many brokers can link together to build a tree. There might be a situation that one broker can saturate that the handlers cannot be supported efficiently. This limitation can be gotten rid of with the introduction of a new broker. Finally, CManager uses efficient publish/subscribe mechanism of NaradaBrokering. Publish/subscribe paradigm is benefited as follows; every computing node has its own topic. In other words, the nodes are uniquely addressable. The messages are sent to those addresses in an order. The topics are mapped with the handlers before communication is started. In a parallel execution, we may assign one topic to the handlers that are concurrently executed. While this reduces the number of addresses in the system, it also prevents modifying the execution flow on the fly. Therefore, we stick the paradigm of a single topic usage for each handler whether it is a parallel or sequential execution.

3.3. Handler Executing Manager

Distributed handlers are the applications executing the messages in remote places. Without having a supportive environment, handlers cannot perform their tasks in remote places. Handler Execution Manager (HEManager) is considered to build this necessary environment. Each distributed handler is hosted by a HEManager. It supports the execution in several ways, stretching out from negotiating with CManager for the communication to creating the necessary structures.

There are as many HEManagers as the number of distributed handlers. Both incoming and outgoing messages travel with DMFormat. When a message arrives to a node, the essential information is extracted and necessary structures are constructed for the handler execution. The structures are built around the unique ID. The execution greatly gets assistance from the properties section of DMFormat. The orchestration is kept hidden from HEManager with the intention of keeping its execution simple. It only knows how to create the environment for the distributed handler and where to send the response.

HEManager leverages the common interfaces to standardize the handler implementation. A handler can be easily implanted to DHArch as far as it implements these interfaces. Moreover, HEManager support some well known handler interfaces such as Apache Axis handler interface.

4. Execution and details

DHArch is a system that is capable of processing Web service handlers in a distributed environment. It is able to use single-processor, multi-processor or multi-core systems as well as facilitates multiple computers sharing a network. It supports parallel as well as sequential execution.

Figure 6 illustrates how a message traversal happens in DHArch. Typically, a message arrives within a context, specifically a Web service container context. The context consists of additional information for the execution as well as the message itself. It also conveys supplementary information about the service requester. Therefore, the incoming message context object is stored completely so that the response to the right place is guaranteed.

DHArch can cooperate with the various Web service containers. 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 one of them. Moreover, conversion between the context objects and DHArch specific common format would be costly. Hence, CMCQueue is utilized to save the interacting container contexts. The context objects are mapped with a unique UUID generated key. Naming is very vital to identify a message. Many messages may arrive to the DHArch in a short duration. The confusion is possible if we cannot differentiate them from each other. Hence, every message has to be uniquely identified to be executed correctly. Otherwise, the execution cannot go through properly because of having confusion in source, destination or processing.

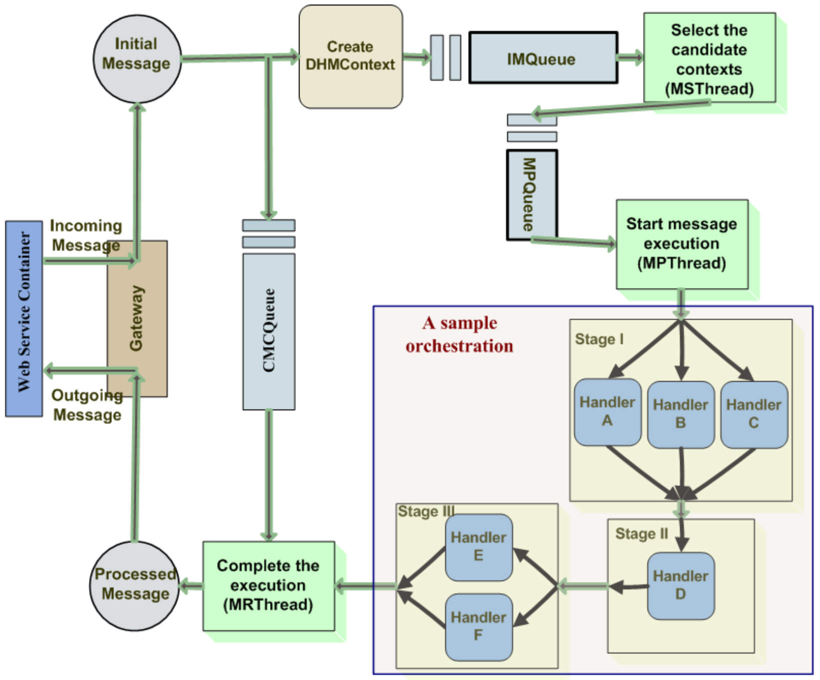

Figure 6 : Message execution

At the same time, DHArch creates its unique message context, DHMContext, to perform its internal execution properly. The container contexts are not utilized for this purpose because of the reason that we want to build an architecture having a container independent execution. Otherwise, we need to revise the execution mechanism for each newly introduced container. DHMContexts are firstly stored in IMQueue. As it is in CMCQueue, IMQueue identifies DHMContext with the same UUID generated unique identifier.

DHMContext employs several structures to contribute the execution. The most important one is the handler orchestration structure. It defines the sequence of the handlers which consists of stages and their corresponding handlers. All basic orchestration constructs are mapped to two simpler processing styles, sequential and parallel. *Stage*s are introduced to support parallel execution. Many stages can be employed in an orchestration structure and their executions are sequential among each other. However, the handlers in a stage are executed concurrently. Each stage should contain at least one handler and there must be more than one handler in a stage to have a parallel execution.

The message processing happens based on the guidance of the orchestration structure. It can be modified during the execution if the orchestration policy allows it. The policy contains the rules about *must* and *mustn't*. A handler orchestration structure may contain several conditions for the correct execution. The policy may dictate the execution sequence. For example, an encryption handler can be forced to be executed first.

When DHMContext is generated and its insertion to IMQueue is completed, the acceptance of the message is finalized. At this moment, the native container context is safe in CMCQueue and DHMContext objects waiting to be selected for the executions are ready in IMQueue. MSThread starts selecting the candidate messages. The candidates are selected according to the First Come First Serve scheme. It is a fair selection because the first arriving message is chosen to be processed first [34].However, the selection scheme can be changed to another queuing scheme such as priority.

MSThread chooses a candidate DHMContext for the execution and places it into MPQueue. This queue is the place where the pipelining happens. There is an optimum value for the number of messages in this queue. Similar management is facilitated in TCP protocol packet rate control procedure[35]. Queue Manager increases the number of contexts in the queue gradually unless the throughput starts diminishing. The optimum value is looked for by increasing and decreasing the number of messages in the queue. Naively, it can 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 that the performance starts deteriorating while the queue size is increasing. This defines the optimum value of the queue size.

MSThread tries to keep MPQueue full. It checks always whether there exist optimum number of message contexts in the queue. If there are enough messages, the thread sleeps. Otherwise, it selects new candidates from IMQueue. MPQueue is much smaller than IMQueue. We have two reasons to employ a smaller queue. The first reason is the message pipelining. The messages are being processed concurrently to allow executing more messages at a time. The second reason is to minimize the access time. The idea is similar to the memory structures of the modern computers; the processes are taken into the caches, smaller and faster memory [37]. Similar to this hierarchical memory structure of the contemporary computers [38], DHArch utilizes a smaller dedicated storage, MPQueue, in addition to the bigger one, IMQueue.

MPThread starts the execution of messages in MPQueue at once. It continues processing messages until the MPQueue becomes empty. While MPThread tries to deplete the messages from MPQueue, MSThread stockpiles new messages on the top of the queue. They work very closely and in tandem style. It is very correct to say that MSThread is a producer while MPThread is consumer.

MPThread carries on the message processing by extracting necessary information from DHMContext. Every distributed handler is located in an addressable place. The addresses are kept within DHMContext. The context also contains the message and the supportive information for the message execution. By using these data, DMFormat is created for the transportation. When it is ready, it is sent to the distributed handlers via CManager. The messages are instantly sent to all handlers of a stage. However, the message execution of those handlers may be completed in different times. All the handler executions in a stage have to be finished before going to the next stage. MPThread waits the completion of the handler executions before starting the delivery of the message to the next stage. This procedure continues until all stages of a message are completed.

DHArch threads require clever notification mechanism. They are not allowed to run continuously. Instead, they are forced to wait if they are not needed. Otherwise, wasting the system resources is inevitable. The threads share the computing resources to be successful in their tasks. The resource sharing happens according to the system thread scheduling algorithm. 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 an actual task [39]. MSThread enters in its wait condition when MPQueue becomes full or IMQueue becomes empty. In both situations, there is really nothing to do for MSThread. Hence, it stays in wait condition until it is notified. There are two notifying events for MSThread. The first one is the number of the messages in MPQueue. If the MPQueue becomes empty or contains fewer messages than the optimum number, MSThread is notified. The second notification is a new message arrival. If MSThread and MPThread are somehow waiting in their conditions, they cannot restart their executions because they notify each other. Therefore, an independent notifier is essential to continue the executions. When a new message arrives to IMQueue and the number of the message in MPQueue is less than the optimum value, MSThread receives a notification.

Handler invocation occurs according to the DHMContext orchestration structure. CManager delivers the messages to their destinations in an order defined by the context orchestration structure. When a message is received by HEManager via CManager, the preparation of the suitable environment for the execution is initiated.

DHArch can utilize wide variety of handlers such as monitoring, format converters, logging, compression, decompression, security, reliability and so on. They generally perform tasks that support to Web service by introducing a new functionality. The interesting part of a SOAP message for a handler is the header even though the body is able to be processed. Therefore, a handler mostly expects the whole SOAP message as an input. On the other hand, many handlers process only the partial SOAP messages. For example, WS-ReliableMessaging handler processes only *wsrm* tag of the entire message. Therefore, HEManager allows utilizing the partial execution where the size of the message becomes a concern. However, since this is not applicable to every handler, a full SOAP message execution is performed unless the partial execution is explicitly mentioned as a necessity. We also need to keep in mind that the partial SOAP message execution causes an overhead originating from parsing the SOAP message and combining the outputs later.

HEManager exploits supplementary data for the handler executions. These data are conveyed within the properties. Some of these properties are applicable to every handler. One of them is oneway feature. It describes a situation that a handler does not have to send any response back. When DHManager encounter an *oneway* handler, it applies fire and forget paradigm and continues its remaining tasks without waiting the response [40]. Additionally, *mustPerform* property is also universal for the handlers. If a handler has true value for the *mustPerform* parameter, it always has to complete its executions. In the situation of an error, the execution has to be repeated if it does not lead to an inconstant state. Otherwise, the message execution must totally be halted and the requester must be informed. The message execution can continue when the *mustPerform* value is false even if the handler throws an exception. For example, skipping a logging handler may not be so crucial for a Web Service so that the message execution can carry on without restarting it from the beginning.

When a handler completes its task, the output message is pushed back to the HEManager. DMFormat is utilized to return the output. The corresponding unique ID has to be same with the request. When it is ready, it is passed to CManager for the delivery to the destination. When the envelope arrives to the destination, MRThread is activated. The message delivery is a notification for MRThread. It updates the corresponding context with the executed message. First, it checks whether the ID is represented in MPQueue. Otherwise, the response is behaved as a malicious message and it is discarded. If the ID passes the check, the properties and the payload are extracted. The corresponding DHMContext in MPQueue is retrieved by using the unique ID. At the end, the context is updated with the processed message.

The modification of a context with a successful handler execution 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 every handler. If it is the case, the context is taken out from the queue. The container context object has been kept in CMCQueue until this moment. It was preserving the essential information to continue the message execution in the interacting Web service container. Therefore, saving the container context object is very important. When the container context is taken out from the CMCQueue, it is updated by utilizing DHMContext that we have retrieved from MPQueue. Finally, the processed container context is passed back to the Web Service container to finalize the message execution in DHArch.

It is possible to have errors while the execution is happening. If a handler stops abruptly because of a failure, the error need to be handled so that the system continues to 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. describes two ways of dealing with failures, *fault prevention* and *fault tolerance* [41]. While the first one works to prevent the occurrence of a fault, the second copes with providing the continuation of the service even in the presence of the failure. Even though a complete avoidance of failure is not possible, there are tools supporting fault prevention [42]. Apparently, fault tolerance is necessary to be able to continue execution while 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 a fault tolerance is mentioned, we need to bear in mind that forward recovery can be used as well as the backward recovery. In the forward recovery, the tasks are tried to be completed by processing several times. Backward capability requires atomicity. It is one of the most essential notions for the consistency. In regard to atomicity, Hagen at al. [43] defines three task types, *atomic*, *quasi-atomic* and *nonatomic*. Atomic tasks are those that they have no effect at all if they fail. For example, every read-only task can be thought as an atomic task even if they fail because it does not cause any change. *Quasi-atomic* effects do not vanish naturally. The effects can be eliminated via a roll-back action, though. *Nonatomic* tasks are the one that the effects cannot be removed when they are committed.

Handlers can be either statefull or stateless. A handler generally processes a SOAP message and applies its procedure over it. In other words, they do not keep any state for the message. This feature contributes to utilizing forward recovery. DHArch restarts the execution if a stateless handler fails. HEManeger notifies the error to DHManager. 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. It 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. In this case, the requester may assume that the handler may be down or crashed.

Handlers are not always stateless. They might be keeping states for the messages. DHArch expects atomicity from the statefull handlers. If a handler fails during its execution, it should not have any effect at all. If having atomic handler is not possible or the handler is a *quasi-atomic,* it is necessary to utilize two-phase commit. There exists a solution for the distributed commit [44]. 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 statefull.

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

5. Measurement and Analysis

We performed series of measurements illustrating the advantages of distributed handler execution in various environments. The first set of measurements is to examine the performance of a single message in DHArch. The second set of experiments is conducted to explore the scalability to illustrate the efficiency of the system. Finally we perform measurements by using two well-known Web Service Specifications, WS-Eventing and WS Resource Framework.

5.1. Performance measurements

DHArch offers a promising environment for Web service handlers. It supports concurrent execution and allows utilizing additional resources. Many types of resources such as computer, processor, memory, storage or even an application can be utilized.

DHArch is evaluated by utilizing 6 different configurations of 5 Web service handlers. They are customized for benchmarking purpose. Two of the handlers are CPU bound handlers. The remaining three handlers have been chosen from the applications that are gradually switching from CPU bound to I/O bound. The handlers are named in these benchmarks as in Table 1. The combinations of the parallel executable handlers can create so many different configurations. However, the dependencies between handlers and the performance issues need to be carefully investigated for the correctness of the execution while deciding these combinations.

Table 1: Handler list for the performance benchmarking

Handler Name	Handler Type
Handler A	CPU Bound
Handler B	CPU Bound
Handler C	IO Bound
Handler D	IO Bound
Handler E	CPU/IO

Out of many, six different handler configurations are selected for this benchmark. The first configuration is sequential execution of the handlers in Apache Axis. This is for the comparison purpose. The second configuration is sequential execution utilizing DHArch. The sequence of the handlers is exactly same with 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 handlers, handler E is executed. The execution time of a handler joining to a parallel execution is significant for the performance. The fourth configuration contains the same number of parallelism. However, in this configuration, the first stage contains Handler A and Handler B and the second stage contains Handler C and Handler D. Therefore, we expect that their performance will be different even though they do the same job. The fifth configuration contains two stages. The handlers are executed concurrently except handler E, which is separated because of the dependency. It modifies the incoming SOAP message. Therefore, it is kept last to prevent an incorrect execution. Finally, the last configuration is created without concerning about the 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. The first environment is a multi-core system. The intention is to figure out the behavior of DHArch in a multi-core system. Nowadays, the trend is to have multi-core computers and it is expected that more cores will be seen in a single processor in near future [45]. Hence, we give a special attention to the measurements in multi-core systems. The utilized machine in this experiment has UltraSPARC T1 processor that contains 8 cores running Solaris Operating System, 4 threads per core, with 8GB physical memory. Although concurrent execution has many challenges [46], it activates the individual core usage in the multi-core systems; a handler may claim its own core. We can conceive this core acquisition as if every handler has its own computing node so that the tasks are achieved without competing for the computing power.

The second benchmarking environment is the computers sharing a Local Area Network. The computers in this cluster have the same hardware features. They utilizes Fedora Core release 1 (Yarrow) in Intel Xeon CPU running on 2.40GHz and 2GB memory. In this environment, the handlers are distributed to the different machines.

The last environment is a single computer, utilizing Pentium 4 processor operating at 2.80GHz with 1.5 GB memory. It is running Red Hat Enterprise Linux AS 4 operating system. In contrast to previous systems, the distributed handlers need to share a single computing resource. Therefore, we may witness context switches among the distributed handlers and the other components of DHArch so that the result may be undesirable for the performance.

5.1.1. Results and analysis

We measure the overall performance of a Web service deployment in Apache Axis and DHArch. Handler distribution causes an overhead. The management of the distributed handler execution and the transportation of tasks increase the execution time. However, there are also gains because of the advantages of parallel execution. Our interest is to find out the performance benefits coming from the advantages of the distribution.

There are ways of compensating the overhead and even achieving a promising overall performance. The first way of improving the performance of a deployment is to establish concurrent handler execution in a distributed environment. Conventional handler deployment does not let the handlers run in a parallel manner. However, there are many independent handlers from each other so that they can process the SOAP messages concurrently. For instance, a monitoring handler does not depend on a logging handler. They can be easily executed concurrently. The second way of improving performance is to utilize faster machines. A faster machine may contribute to the overall performance when an appropriate handler is deployed into it. For instance, encryption and decryption handlers' distribution to the faster machines within a secure environment contributes best to the overall system.

The measurement, shown in Figure 7, 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 execution times of the handlers. Every observation was repeated 100 times.

It is clearly seen that the best results are observed when all handlers are able to run concurrently. However, processing them concurrently may not be always possible. As we discussed earlier, the dependencies between the handlers have to be considered. For example, an encryption handler may need to be processed first. Otherwise, the remaining handlers cannot understand the message because of the encryption. The difference between configuration 1 and 2 is the overhead originating from the distribution of five handlers. The first configuration utilizes Apache Axis in-memory handler deployment. In the second configuration, because of the distribution of the handlers to the individual cores, DHArch increases the execution time slightly. The gain may be small; in the configuration 3, it is around 50-70 milliseconds because of the processing time of Handler C and Handler D. As a result, this configuration slightly provides enough gain to overcome the overhead. Sometimes, gain may not even compensate the overhead. On the other hand, a gain can be very appealing as it is in configuration 4, 5 and 6. They provide good results due to processing times of Handler A and Handler B. The numerical values of the results are stated in Table 2.

Gain completely depends on handler configuration. On the one hand, it can provide a fascinating performance with the execution of all the handlers in a parallel manner. On the other hand, it may not even present a slight gain in the execution time to compensate the overhead coming from the distribution of handlers.

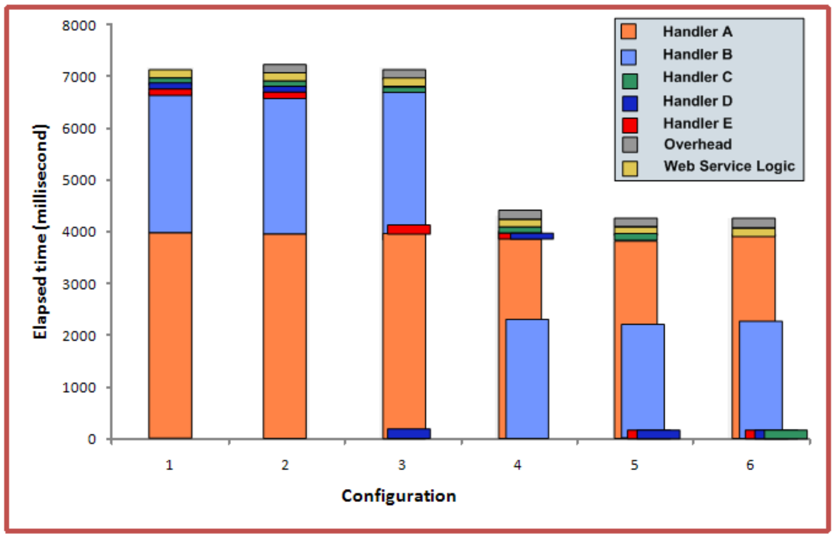

Figure 7: The execution of Web service containing the five handlers with six handler configurations in the multi-core system

Table 2: The elapsed time and the standard deviation of the performance benchmark in the multi-core system

Configuration number	1	2	3	4	5	6
Mean value (msec)	7192.9	7220.92	7164.98	4324.86	4279.37	4264.78
Standard deviation	42.97	56.68	57.75	49.66	29.92	36.96

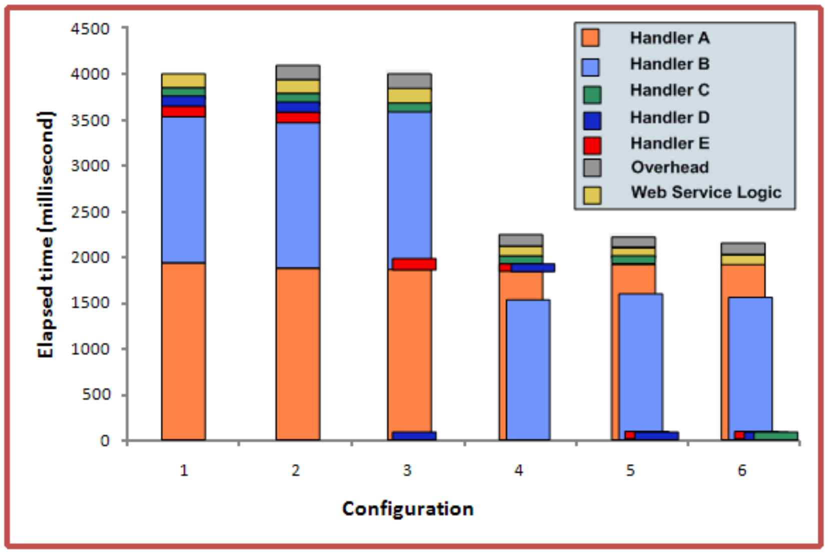

Figure 8 : The execution of Web service containing the five handlers with six handler configurations in the cluster utilizing Local Area Network

Figure 8 illustrates results from the executions of the handlers in cluster that communicates with a Local Area Network. 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 previous systems. The sequential execution of DHArch is executed slower than those from the remaining configurations. The numerical values of the results are shown in Table 3.

The standard deviations are reasonable even if the tasks between handlers travel over the local network. The network is fast and consistent. The message transportation does not take too much time. When the results are compared with those from the previous systems, any side effect coming from the usage of LAN is not observed.

Table 3: The elapsed time and the standard deviation of the performance benchmark in the cluster utilizing Local Area Network

Configuration number	1	2	3	4	5	6
Mean value (msec)	1717.08	1741.95	1712.22	1182.06	1150.55	1139.26
Standard Deviation (msec)	42.56	35.32	36.30	44.06	37.79	45.90

The results, shown in Figure 9 and Table 4, are from the single processor system. In contrast to previous measurements, single processor system provides a different pattern. Thread scheduling becomes an issue. Since two handlers are heavily CPU-bound, the individual execution times of them are increasing when they are executed concurrently. Moreover, NaradaBrokering and Apache Axis in Apache Tomcat container use the same processor. This worsens the thread scheduling.

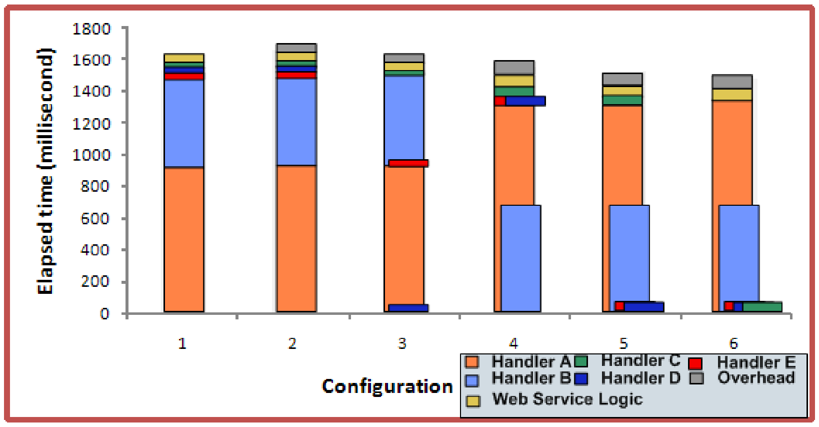

Figure 9 : The execution of Web service containing the five handlers with six handler configurations in the single processor system

Table 4 : The elapsed time and the standard deviation of the performance benchmark in the single processor system

Configuration number	1	2	3	4	5	6
Mean value (msec)	1538.14	1661.73	1638.54	1558.9	1528.21	1488.67
Standard Deviation (msec)	56.32	58.29	54.86	73.82	85.90	86.80

5.2. Scalability

In this experiment, we will investigate the throughput in DHArch comparing with a conventional handler mechanism. We will also find answers for the effect of request rate over the processing time.

5.2.1. Message rate

Web service is basically a paradigm that clients make requests to execute a task in a remote application. This structure may lead the situation that many clients make requests in a short time. For instance, an online shopping center which utilizes Web service technology may receive hundreds of transactions. There might be scenarios that the request rate may be even higher. For example, Web service, which presents an interface to illustrate a real time tornado development, may receive inputs from thousands of sensors. Consequently, a Web service may have a very high request rate. Therefore, the system architecture must be efficient and effective that it can answer the increasing number of requests. Handler chain is one of the most crucial parts of the service execution. Its performance directly affects overall system performance. We will investigate the scalability of DHArch by comparing with Apache Axis 1.x handler execution mechanism.

We utilize multi-core machines in a cluster for benchmarking purpose. In this cluster, 8 computers communicate via a Local Area Network. Every computer has 2 Quad-core Intel Xeon processors running at 2.33 GHz with 8 GB of memory and operating Red Hat Enterprise Linux ES release 4 (Nahant Update 4).

Three handlers are utilized for this measurement: 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, and information about the clients, and number of clients which are connected and so on. The last handler, Format Converter, converts incoming message formats to a uniform format, a format that the service expects.

Apache Axis handler structure utilizes a cain of handlers that passes the massage from one handler to another. A configuration file defines the handlers and their position in the execution path. Apache Axis handler executions can be depicted as it is in Figure 10.

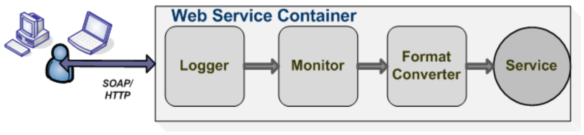

Figure 10: Apache Axis sequential handler deployment for scalability measurement

DHArch provides concurrent as well as sequential execution for the handlers. The group of handlers, used in Apache Axis benchmarking, is also utilized for DHArch. Even though handlers may not be possibly processed in a parallel manner because of dependencies, the handlers selected for this experiment are suitable for parallel execution. Hence parallel as well as sequential execution has been utilized for DHArch benchmarking. The deployments are portrayed in Figure 11 and Figure 12.

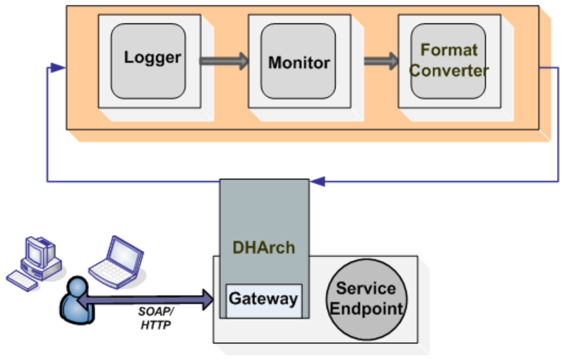

Figure 11: DHArch sequential handler deployment for the scalability measurement

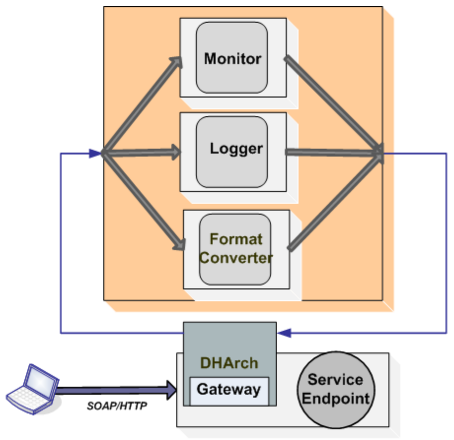

Figure 12: DHArch parallel handler deployment for the scalability measurement

Two experiments have been conducted. In the first one, we have measured elapsed execution time of a single message while the number of messages per second is increasing. The second experiment has been performed to measure the cumulative time for the completion of the certain number of messages.

In order to measure the execution time for a single message while the number of message per second is increasing, we use the following experimental setup. The messages are sent within the same rate during 100 seconds. The rate starts from 1 message per second and continually increases 10 messages in every step to the level that the service can support. Figure 13 shows the results gathered from a single machine in the cluster. A single machine is used for the comparison purpose. The figure shows the elapsed execution times, measured in client side.

DHArch parallel execution has the fastest execution time while the sequential execution yields the highest processing time because of the distribution. Between these two, we see Apache Axis result. At one point, the processing times increases noticeably. This incident happens where the system resources are fully utilized. The message execution time has been slowly increasing because every additional message starts sharing the computing resources. However, it has not been causing abrupt changes until the resources are fully used. When the resources start unable to meet the demands, the execution times has been skyrocketing. In Apache Axis, every arriving message starts another handler pipelining which shares the scarce resources. The context switches starts occurring more frequently. Hence, the execution time increases faster. There is not a regulation for the incoming messages to prevent this dilemma. On the other hand, DHArch has a different reason for the spike. DHArch does not allow the context switching cost worsening the system performance. Instead, the increase in execution time comes from the message waiting in IMQueue. DHArch forces the messages wait in IMQueue; it keeps optimum number of messages in MPQueue not to worsen the processing time because of the context switching. Hence, we observe a slower increment in the execution time for the message rate between 70 and 80.

For the Apache Axis deployment, we observe that the message execution time started to decline significantly when the number of threads hits a point that thread scheduling becomes an issue. The performance begins deteriorating dramatically. The problem is that there are too many threads running and handler mechanism did not have any regulation to keep the performance in its optimum level. We notice that the fluctuation in the message processing increases considerably. When the engine completes enough message executions, the performance is improving and the system starts processing more messages. At the same time, the newly arriving messages begin building up the new threads. When it reached its limits, the performance starts declining again. This pattern repeats itself until the message executions are completed. The standard deviation for 80 messages per second illustrates the incident. On the other hand, since context switching does not affect the execution as it is in Apache Axis, the same fluctuation is not observed in DHArch. However, the increase in the execution time is not preventable when the system resources are drenched. In order to optimize message execution, the remaining messages that system cannot support are forced to wait in the queue. Hence, the message processing time increases steadily in DHArch.

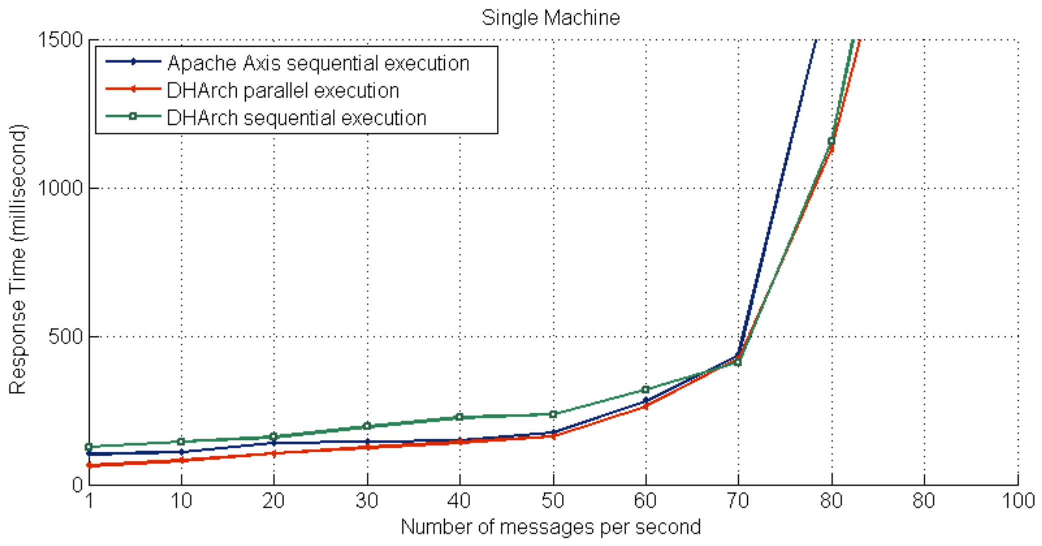

Figure 13 : Message execution for increasing message rate in a single machine

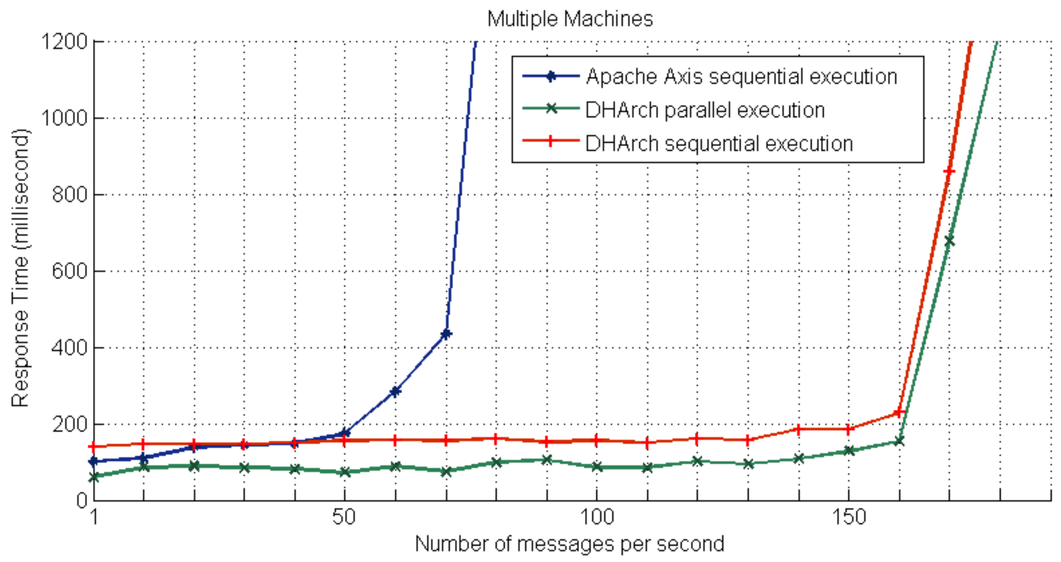

Figure 14 : Message execution for increasing number of messages per second in multiple machines communicating via Local Area Network

A multi-core system provides advantages due to the individual core use for handler executions. If the resources are enough for the handlers which are running in a parallel manner, the computing resources do not have to be relinquished while the execution continues. For a single request, we definitely see the advantage of utilizing individual cores when the handler parallelism is applied. On the other hand, the advantage of the parallel execution of the handlers fades away for higher message rates. In other words, pipelining becomes dominant factor in the executions. Both Apache Axis and DHArch benefits from pipelining. Hence, in this experiment, we investigate mainly pipelining rather than handler parallelism.

When we introduce multiple computers, we see the immense gain in DHArch. Apache Axis cannot benefit from multiple computers but DHArch can. Hence, the processing time stays stable longer time. Figure 14 portrays this situation. The message rate does not change the response time until 160 messages per second. One of the important events in the graph is the convergence of the Apache Axis single machine execution to the DHArch multiple machine sequential execution. In a single machine, Apache Axis processes massages faster than DHArch sequential execution. When we introduce the additional computers for DHArch, Apache Axis catches and later passes the execution time of DHArch sequential.

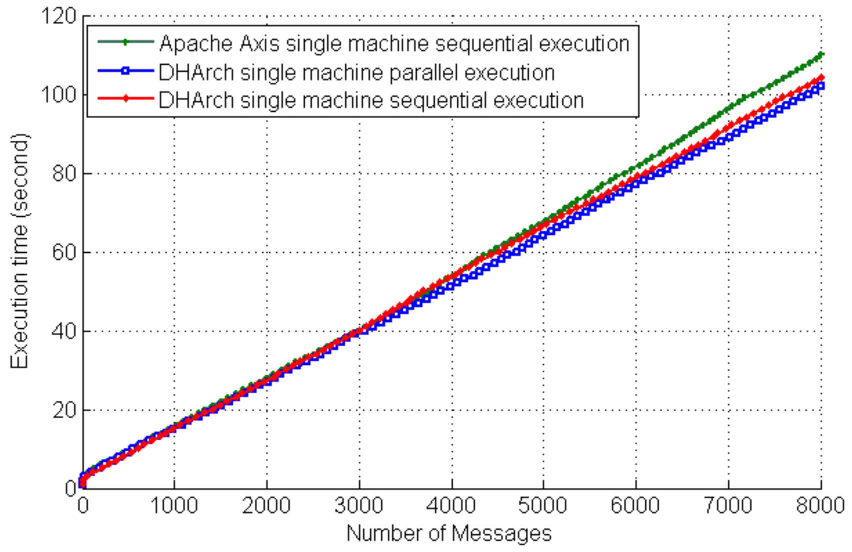

Figure 15 : Execution for increasing number of messages in a single machine

In the second experiment, message rate is 80 messages per second where the system resources start being utilized fully in a single machine. The message rate is kept same for 100 seconds. In other words, 8000 messages are sent in total. In every second, we measure the cumulative number of the executed messages. The results are depicted in Figure 15.

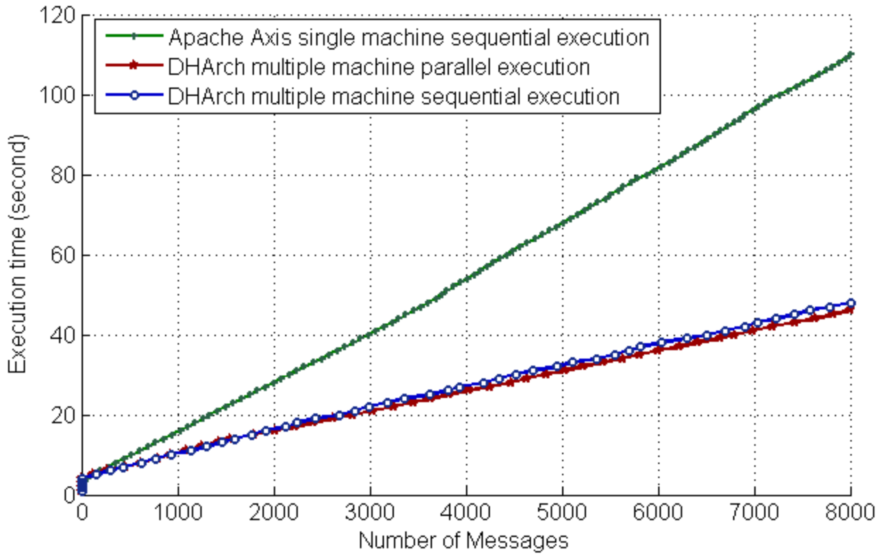

Figure 16 : Execution for increasing number of messages in multiple machines

When we look at the graph, we notice that Apache Axis completes its executions later than DHArch. The reason is the thread scheduling. DHArch employs a regulatory mechanism to control thread scheduling. It does not allow creating too many parallel message execution pipelines that shares the resources and causes performance degradation. Another observation from the figure is the closeness of the parallel and sequential executions of DHArch. While the system resources are being used fully, the parallel or sequential execution does not differ so much because the dominant factor is pipelining rather than handler parallelism.

When additional computers are introduced to DHArch, the performance becomes very promising. The processing time of the same amount of messages is reduced more than two fold and number of messages executed in a given time is increased considerably. Figure 16 portrays the results.

We clearly notice the advantages of utilizing DHArch in terms of throughput when multiple computes are used for the computation, shown in Table 5. In single machine, the message rate is 80 messages per second. The throughputs are very close to one another. When the multiple machines are used in DHArch, the throughput becomes favorable to the DHArch because the number of the processed messages doubles.

Table 5 : Throughput where the system resources are being utilized fully

	Throughput (messages per second)
Apache Axis in single machine	72
DHArch sequential in single machine	78
DHArch parallel in single machine	76
DHArch sequential in multiple machines	166
DHArch parallel in multiple machines	173

5.3 Deploying Web Service Resource Framework and Web Service Eventing

We want to crown the experiments by showing deployment of two well-known Web service specifications. Many efforts have been dedicated to the WS-specification. The implementations gradually have started to appear Web service arena. We have found several groups providing the WS-specification implementations. Among them, two specs were fitting for our purpose; WS-Resource Framework [32] and WS-Eventing[47].

Web services must offer ability to the clients to access and manipulate state. Even though managing states is challenging, stateful resources are not utterly evitable. A service may utilize one or more stateful resources. Hence, Web service architecture should provide eligible functionalities to access them. On the other hand, while this capability is being offered, having a standard way is essential. Web Service Resource Framework (WSRF) establishes the necessary standards for the states. It provides capabilities to insert, update, and discover the stateful resources in a standard and interoperable way. We utilize the Apache implementation of WSRF for the experimental purpose. We created our stateful resource for *sensors*. In addition to inquiry, insert and update functionalities can also be achieved in a standard way.

A Web service may benefit from receiving a 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 as subscriber, needs to register itself to a certain interest with another service, called as event source. Web Service Eventing (WS-Eventing) defines a protocol to standardize this effort. A subscription manager can be employed to administer subscriptions. We utilize FIN, an implementation of WS-Eventing from Pervasive Technology Lab [48].

A computer cluster is utilized for this experiment. It contains 8 machines having the same features. They share Local Area Network to communicate each other and utilize Fedora Core release 1 (Yarrow) in Intel Xeon CPU running on 2.40GHz and 2GB memory.

Before starting benchmarking, the initializations of the specifications are completed. Sink registers itself to the topic /sensor/california and sensor stateful resource stores the initial information. Most importantly, the suitable massages are selected, one from WS-Eventing and one from WSRF. These messages are combined to create a new message in order to run WSRF and WS-Eventing handlers in a parallel manner.

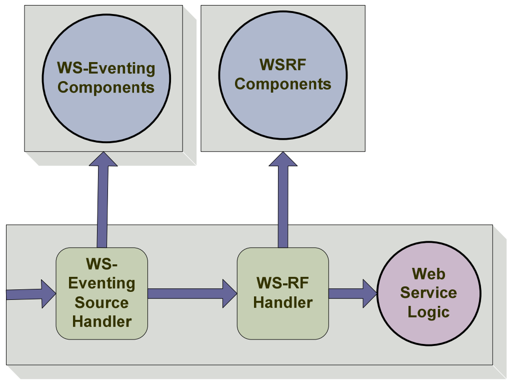

Figure 17 : Sequential Execution of WSRF and WS-Eventing

The message notifies an important activity and updates information for a sensor stateful resource. When it is received, WS-Eventing source handler looks for the subscription manager service to find out the interested subscribers. Then, it delivers the event to the sinks, the interested subscribers. While notification is happening, WSRF handler also updates the values of the states, which are kept in storage, and forwards the information with the additional data previously stored.

Specifications are, first, deployed for Apache Axis. Every single request is observed 100 times. Handlers and service endpoint utilize a single computer. The remaining components of the specifications are hosted by the individual computers in the cluster. The logical deployment is depicted in Figure 17.

The environment to execute WS- Eventing and WSRF is also created for DHArch.WS-Eventing requires individual computers for its components; Sink Source and Subscription Manager. Hence, they are located to the separate computers. Similarly, WSRF as well as NaradaBrokering are located into the individual computers in the cluster. Finally, the service endpoint is placed its location in the cluster. The deployment can be portrayed as in Figure 18.

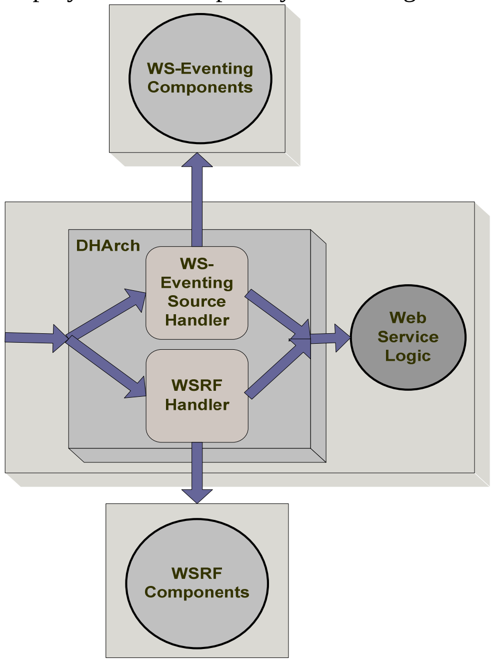

Figure 18 : Parallel Execution of WSRF and WS-Eventing

Table 6 : WSRF and WS-Eventing sequential execution in Axis handler structure

	WSRF	WS-Eventing	Total service
Execution time (millisecond)	69.32	55.08	162.14
Standard deviation	6.51	4.98	7.18

We first gathered the results in Apache Axis by running WS-Eventing and WSRF sequentially. The handlers are deployed into the request path. They look for their responsible elements in the messages. In other words, the handlers process only the relevant elements. We have individually measured the execution times of the WSRF and WS-Eventing. The results are shown in Table 6.

We perform the same sequential handler execution in DHArch. Because of the overhead originating from the distribution of the handlers, the time of processing a single message increases. The results are shown in Table 7.

Table 7 : WSRF and WS-Eventing sequential execution in DHArch

	WSRF	WS-Eventing	Total service
Execution time (millisecond)	70.25	54.68	171.64
Standard deviation	4.45	3.93	10.08

When we introduce the parallelism, we see significant improvement in the service performance. The concurrency reduces the execution cost of a single request by one forth. The cumulative execution time of the handlers in a sequential processing is around 124 milliseconds. It is slightly higher than the total execution time of the service in DHArch parallel handler execution. Since WSRF processing time is higher, it is the main player to determine the processing time of the handlers joining to the parallel execution. Due to the fact that DHArch deals with only handlers, the service endpoint processing time does not change. A service without handler executions takes almost 40 milliseconds. Table 8 shows the execution times and standard deviations of DHArch parallel handler execution.

Table 8 : WSRF and WS-Eventing parallel execution in DHArch

	WSRF	WS-Eventing	Total service
Execution time (millisecond)	69.49	54.45	115.15
Standard deviation	5.53	3.42	12.15

The benchmarking demonstrates the advantage of parallelism for the handler execution. While the search goes on for the handler candidates among the specification, we encounter a very small domain of handlers which is possibly executable concurrently. Even in this domain, the way of implementation causes problems for the distribution. We are expecting that this domain grows in near future. Hence, utilizing the distribution and parallelism for the specifications will produce many state-of-art applications.

6. Conclusion

Service Oriented Architectures, specifically Web service technologies, focus on benefiting maximally from interoperability and reusability.

Many standards and structures have been developed to provide an interoperable environment. Web Service Description Language (WSDL), Universal Description Discovery and Integration (UDDI) and Simple Object Access Protocol (SOAP) are de-facto standards to build Web services, which are basically an application offering a service via SOAP messaging. On top of these standards, many WS-specifications have been introduced to provide additional capabilities. Many others are already on the way. Furthermore, there are efforts to build efficient Web service processing environments. These environments contain many tools to process SOAP messages, which is the most basic and essential task of a service execution framework. Hence, SOAP processing engines, Web service containers, have been constructed to provide an efficient environment and to hide the complexity of the SOAP processing from the user.

Web services exploit additive functionalities to improve its capabilities such as security, reliability, logging and so on. Some of these functionalities have been standardized as WS-specifications such as WS-Security and WS-Reliable Messaging. In many cases, the functionalities are very essential for a service. For example, a health service without reliability may be deadly. A monitoring service without logging may be useless. Consequently, a Web service needs additional functionalities to improve its capabilities. These additive functionalities are called handlers or filters. They are inevitable for many services as the necessary capabilities are stated for the health and monitoring services. This necessity forces the containers to create their internal handler architecture. However, the design is very critical in order to be successful in this effort. Since handlers are one of the key SOAP processing component of Web service architecture, this design affects the whole Web service execution. Therefore, we have investigated the handler architectures extensively and derived very vital and important results from this conclusive research. Distributed Handler Architecture (DHArch) shows us many essential features that are necessary for efficient, scalable, flexible, and modular handler architecture.

DHArch provides very efficient handler architecture by exploiting concurrent handler execution and utilizing additional resources. Many handlers are independent from each other. In other words, they can be processed concurrently without harming the correctness of the execution. This improves the performance dramatically. Moreover, the efficiency significantly increases when the parallel executions leverages additional resources. For example, taking advantage of an individual powerful machine for WS-Security in LAN network contributes to the system efficiency incredibly.

DHArch benefits from message parallelism in addition to the handler parallelism. Instead of waiting for the completion of a message execution, many messages can be processed at the same time. We called this *message pipelining*. DHArch utilizes pipelining by leveraging its internal structures. DHArch processes the optimum number of messages and keeps the remaining in a queue instead of letting every message arriving to the system to start its execution right away. This regulation prevents the performance degradation because of too many messages running concurrently.

Orchestration is a significant feature to collaborate the distributed applications. Dissemination of the handlers requires a handler orchestration. Promising results cannot be expected without a decent orchestration mechanism for the handlers. Hence, an orchestration mechanism has been introduced. It provides two main advantages. First of all, it offers very efficient and effective engine by introduction of the separation of the description and the execution while it is providing very powerful expressiveness. Secondly, this mechanism helps to build dynamic handler structure.

DHArch scales very well. Having additional resources improves the scalability. More resources allow answering more requests. Since a Web service may contain many handlers in addition to the Service endpoint, they all together may saturate a single machine. It gets worse while many clients are requesting many services concurrently. The response time keeps increasing. Instead, the bottleneck points can be eliminated by introducing additional resources and utilization of the concurrency.

DHArch is a very flexible system. It easily allows adding new handlers. The architecture can also easily be adapted to a Web service container. The only necessary action is the implanting a suitable gateway. Furthermore, it is also able to utilize a variety of platforms for the handler distribution. It can process handlers in a system ranging from a single computer, multi-core, and multi processor to many computers.

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. Apache Axis, http://ws.apache.org/axis/.
6. Microsoft Web Service Enhancements (WSE), http://www.microsoft.com/downloads/details.aspx?FamilyId=FC5F06C5-821F-41D3-A4FE-6C7B56423841&displaylang=en.

7. IBM WebSphere, http://www-306.ibm.com/software/websphere/.
8. Web Service Specifications, http://www-128.ibm.com/developerworks/webservices/library/ws-spec.html.
9. Hoare, C.A.R., *The emperor's old clothes*. 1981, ACM Press New York, NY, USA. p. 75-83.
10. Perera, S., C.Herath, J. Ekanayake, E. Chinthaka, A. Ranabahu, D. Jayasinghe, S. Weerawarana, G. Daniels, *Axis2, Middleware for Next Generation Web Services.* . in IEEE International Conference on Web Services (ICWS'06). 2006.
11. Web Service Addressing(WS-Addressing), http://www.w3.org/Submission/ws-addressing/.
12. Shrideep Pallickara, et al., *On the Costs for Reliable Messaging in Web/Grid Service Environments.* Proceedings of the 2005 IEEE International Conference on e-Science & Grid Computing. Melbourne, Australia.pp 344-351.
13. Shirasuna, S., et al., *Performance comparison of security mechanisms for grid services*. p. 360-364.
14. Slominski, A., et al., *Asynchronous Peer-to-Peer Web Services and Firewalls*, In 7th International Workshop on Java for Parallel and Distributed Programming (IPDPS 2005), April 2005.
15. Fang, L., A. Slominski, and D. Gannon, *Web Services Security and Load Balancing in Grid Environment.*
16. Beytullah Yildiz, Geoffrey Fox, Shrideep Pallickara *An Orchestration for Distributed Web Service Handlers* The Third International Conference on Internet and Web Applications and Services ICIW 2008 June 8-13, 2008 - Athens, Greece
17. Fox, G., Pallickara, S., and Parastatidis, S, *Toward Flexible Messaging for SOAP-Based Services.* In Proceedings of the 2004 ACM/IEEE Conference on Supercomputing (November 06 - 12, 2004). Conference on High Performance Networking and Computing.
18. Arulanthu, A.B., et al., *The Design and Performance of a Scalable ORB Architecture for CORBA Asynchronous Messaging* in Proceedings of the Middleware 2000 Conference, ACM/IFIP, Apr. 2000.
19. L. Bellissard, et al., *An Agent Platform for Reliable Asynchronous Distributed Programming* In Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (October 18 - 21, 1999).
20. Langendoen K., R. Bhoedjang, and H. Bal, *Models for Asynchronous Message Handling* IEEE Parallel Distrib. Technol. 5, 2 (Apr. 1997), 28-38.
21. Buchmann, S.K.A., *Improving Data Access of J2EE Applications by Exploiting Asynchronous Messaging and Caching Services.* in Proceeding of the 28th International Conference on Very Large Data Bases; 2002
22. Thakur R., W. Gropp, and E. Lusk, *On implementing MPI-IO portably and with high performance.* In Proceedings of the Sixth Workshop on I/O in Parallel and Distributed Systems (Atlanta, Georgia, United States, May 05 - 05, 1999). IOPADS '99. ACM Press, New York, NY.
23. Amir Y., et al., *A cost-benefit flow control for reliable multicast and unicast in overlay networks.* IEEE/ACM Trans. Netw. 13, 5 (Oct. 2005), 1094-1106.
24. Shenker, S., *A theoretical analysis of feedback flow control.* In Proceedings of the ACM Symposium on Communications Architectures &Amp; Protocols (Philadelphia, Pennsylvania, United States, September 26 - 28, 1990). SIGCOMM '90. ACM Press, New York, NY.
25. Shivers, O., *Control flow analysis in scheme.* In Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation (Atlanta, Georgia, United States, June 20 - 24, 1988). R. L. Wexelblat, Ed. PLDI '88. ACM Press, New York, NY.
26. Qiu D. and N.B. Shroff, *A new predictive flow control scheme for efficient network utilization and QoS.* In Proceedings of the 2001 ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems (Cambridge, Massachusetts, United States). SIGMETRICS '01. ACM Press, New York, NY, 143-153.
27. Fox, G., *2004. A Scheme for Reliable Delivery of Events in Distributed Middleware Systems.* In Proceedings of the First international Conference on Autonomic Computing (Icac'04) - Volume 00 (May 17 - 18, 2004). ICAC. IEEE Computer Society, Washington, DC, 328-329.
28. Tai, S., Thomas A. Mikalsen, and Isabelle Rouvellou, *Using Message-oriented Middleware for Reliable Web Services Messaging.* Lecture notes in computer science (Lect. notes comput. sci.) ISSN 0302-9743 , 2003.
29. S Maffeis and D.C. Schmidt, *Constructing Reliable Distributed Communication Systems with CORBA.* IEEE Comm., Feb. 1997.
30. Web Service Reliable Messaging (WS-ReliableMessaging), ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200502.pdf.
31. Apache WSRF, An implementation of WS-Resource Framework, http://ws.apache.org/wsrf/.
32. Web Service Recource Framework (WS-Recource Framework), http://docs.oasis-open.org/wsrf/wsrf-primer-1.2-primer-cd-02.pdf.
33. Web Services Reliability (WS-Reliability) http://www.oracle.com/technology/tech/webservices/htdocs/spec/WS-ReliabilityV1.0.pdf.
34. van Emde Boas, P., R. Kaas, and E. Zijlstra, *Design and implementation of an efficient priority queue*. 1976, Springer. p. 99-127.
35. Handley, M., et al., *RFC3448: TCP Friendly Rate Control (TFRC): Protocol Specification*. 2003, RFC Editor United States.
36. Transmission Control Protocol (TCP) http://www.ietf.org/rfc/rfc793.txt.
37. Avid Karger , A.S., Andy Berkheimer , Bill Bogstad , Rizwan Dhanidina , Ken Iwamoto , Brian Kim , Luke Matkins , Yoav Yerushalmi, *Web caching with consistent hashing.* Proceeding of the eighth international conference on World Wide Web, p.1203-1213, May 1999, Toronto, Canada
38. Aggarwal , B.A., A. Chandra , M. Snir, *A model for hierarchical vmemory,* Proceedings of the nineteenth annual ACM conference on Theory of computing, p.305-314, January 1987, New York, New York, United States
39. Abraham Silberschatz, G.G., Peter Baer Galvin, *Operating system concepts*. 2002: Addison-Wesley Reading, Mass.
40. Voelter, M., M. Kircher, and U. Zdun, *Patterns for asynchronous invocations in distributed object frameworks*, EuroPLoP 2003, http://www.kircher-schwanninger.de/michael/publications/AsynchronyEuroPLoP2003.pdf.
41. Laprie, J.C.C., A. Avizienis, and H. Kopetz, *Dependability: Basic Concepts and Terminology*. 1992, Springer-Verlag New York, Inc. Secaucus, NJ, USA.
42. Leymann, F. and W. Altenhuber, *Managing Business Processes an an Information Resource*. 1994. p. 326-348.
43. Hagen, C. and G. Alonso, *Exception handling in workflow management systems*. 2000. p. 943-958.
44. Gray, J., *Notes on Data Base Operating Systems*. 1978: Springer-Verlag London, UK.
45. Johnson, C.a.W., J *Future processors: flexible and modular*. In Proceedings of the 3rd IEEE/ACM/IFIP international Conference on Hardware/Software Codesign and System Synthesis (Jersey City, NJ, USA, September 19 - 21, 2005). *CODES+ISSS '05. ACM Press, New York, NY, 4-6.* 2005
46. Majumdar, S., Eager, D. L., and Bunt, R. B. , *Scheduling in multiprogrammed parallel systems.* SIGMETRICS Perform. Eval. Rev. 16, 1 (May. 1988), 104-113. 1988.
47. Web Service Eventing (WS-Eventing), http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf.
48. FINS, An Implementation of WS-Eventing, http://www.naradabrokering.org/FINS-Docs/.