Configuration Management in Multi-Channel Multi-Container Web Application Servers

Kareim M. Sobhe and Ahmed Sameh

Abstract—**The Multi-Container Web server allows the division of a single web request into independent portions to be executed in parallel over different communication channels. To achieve this, the underlying communication infrastructure of traditional web environments is changed from the state-full TCP to the stateless UDP communication protocol. Such an architectural change provided an environment suitable for parallelization and distribution and enhanced some other already existing characteristics of the current web environments such as fault tolerance and web caching. In this paper, performance is further enhanced as part of the framework upon which web applications will run by introducing a number of changes. The configuration Manager (CM) is the subsystem that is responsible for reading configuration files provided by administrator. The CM is split into two parts, which are the Container CM and the HPA CM. After the CM reads the necessary configuration files and validates its content, it makes it easy for the administrators to be able to change environment setting such as container port ranges, types of nodes, services names, services location, number of instances each service should be loaded, and more. The CM also have the capability to reconstruct some types of configuration files from its data structure buffers, as such buffers may change through other subsystems and such changes may need to be permanent.**

Index Terms—**Multi-Channel, Clustering, High Availability, Service State Migration, High Performance Computing, High Performance Agent, Skeleton Caching, Containers, Hybrid Scripting**

I. Introduction

In previous papers we proposed a web application server [1][2][3] that is an application deployment server to load application component instances in the form of services, as well as provide the resources required for them to execute and function. The web Container supports a UDP based communication layer through which all communication between any Container and its clients are over a state-full communication protocol built on top of UDP. Since a Container will not be able to communicate except through a proprietary protocol based on UDP, and since normal web clients communicate with web servers using HTTP over TCP, an intermediate translator is necessary to narrow the gap and enable the web client to transparently send its requests to the container. Thus, the High Performance Agent component is introduced which will be referred to throughout this paper as HPA. Acting as a reverse proxy, the HPA is located physically on the machine which the web client initiates its web requests from. Unlike any normal proxy, the HPA provides proxy operations between a web client and a Container over different communication protocols, so the HPA will be communicating with the web client through normal HTTP over TCP and will translate those client requests to the container through an extended HTTP protocol over UDP. The HPA is designed to be a reverse proxy because unlike normal proxies, a reverse proxy serves a specific destination or a number of destinations. In a realistic situation, the HPA is not considered an overhead, as it is located on the client machine, very tightly coupled with the web client and serves only the normal load of a single user's web transactions. Figure 1 shows the proposed new architecture. The Container is a normal web application server deployment container with all the subsystems needed to carry out the basic functionalities of a normal web application server deployment container which are loading application business logic components in the form of loadable services components, and providing them with the necessary resources to be able to operate and function. The Container has a class hierarchy that any service needs to extend to be able to be deployed in the Container. Services should be developed and implemented in the development technology that a container supports; in this case, the proposed environment will support hybrid development and runtime technology types of containers which will all be replicas in architecture and provide the same communication interface, so there will be C++ Containers and Java Containers. Maybe in the future there will be PERL containers, PHP Containers, Python Containers, ...etc., where the responsibility of each container type is to host services that are developed with its supported technology in mind, for example, the C++ container will host services developed in C++. As will be seen in the next sections, a web request could be broken down to portions that may run on different development technology container nodes, and those hybrid services can exchange messages.

A container node has a multi-thread communication layer with pre-allocated communication sockets to communicate concurrently with different clients. A service factory is required to load service instances in ready-to-execute threads to assign to service requests coming from the clients. The service factory loads services that are defined in the container configuration files, thus a configuration manager subsystem is needed to parse and load configuration files which define the settings that the container should have such as the communication port range that the container should acquire, maximum number of communication threads, services names that the container should load,

Manuscript received April 29, 2010; revised February 8, 2011.

Kareim M. Sobhe is with the Department of Computer Science, Ahmed Sameh Department of Computer Science.

The American University in Cairo, Prince Sultan University Cairo, 11511, Egypt, Riyadh, 66833, Saudi Arabia Sameh.aucegypt.edu@ gmail.com.

number of instances to be instantiated from each service type, location of multi-channel server side scripts called skeletons, ...etc.

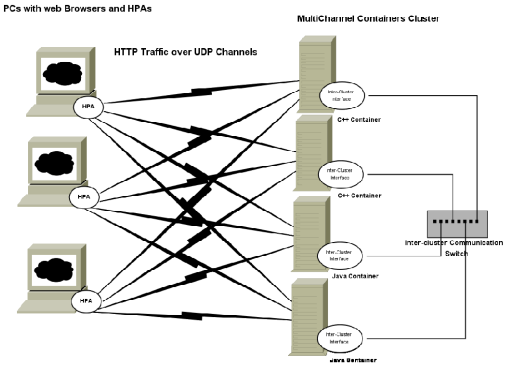


Fig 1. The Proposed Multi-Channel Web Environment based on UDP

The container node has a dispatcher which dispatches incoming web transactions to the correct services to handle the request, and also a communication buffer manager to assign and manage communication buffers allocated for dispatched services. As can be seen, many resources are allocated by a container node such as communication threads, communication buffers, memory and thread resources for instantiated services instances, therefore a garbage collector is needed for environment housekeeping for expired resources to enable them to be reinitialized and reused for following requests. Each component will have its own garbage collection module. For example the factory will be able to clean and reacquire terminated service instances after they finish execution. The communication layer will be able to clean up finished communication channels and reinitialize them for further reuse. The communication buffer manager will be able to de-allocate expired unused communication buffers. Figure 2 shows the internal architecture of a container node irrespective of its supported development and runtime technology.

So far, the architecture presented serves single container functionality, so a cluster management subsystem will be added to enable message exchange between different container nodes which will help in the proposed multichannel mechanisms and through which service state migration, which is discussed later, will provide a better infrastructure for fault tolerance. To deploy services easily, a deployment manager subsystem will work closely with the cluster management subsystem to enable the clustered deployment of services which will include service images replication on container clustered nodes. In fact, the deployment manager will use the cluster management subsystem's APIs and interfaces to carry out cross cluster deployment operations Each Container type has an embedded class hierarchy all of which follow the same design and functionality as much as possible. For a service to be deployed in a specific container it should extend an Ancestor Class which is provided by all container types. The Ancestor class basically has two sets of methods; the first set is those methods which are basic virtual methods for services to extend and overload such as the main method which is called by the service factory when a service is dispatched to serve a web request.

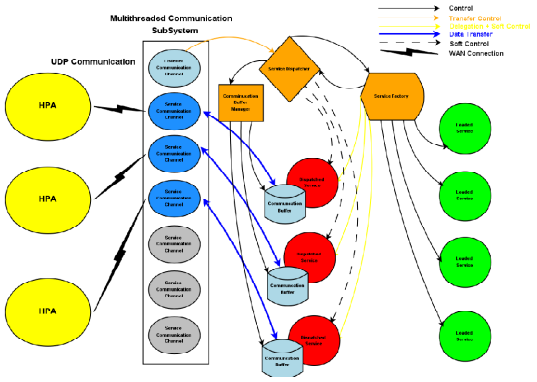


Fig 2. Container Node Architecture

The other set of methods has the role of encapsulating functionalities that are carried out by the container on behalf of services such as reading and parsing the HTTP Request header and posted data as well as composing the HTTP Reply header. It is very important that the service developer be aware of the container class hierarchy and its interfaces to be able to utilize the container functionalities and its internal infrastructure.

The container can serve two types of services which are designed to enable the developer of application components to develop applications in a decomposable way that will enable the concurrent execution of services, and the delivery of their results over multiple communication channels: 1- Single Channel Services: The first type of services is the Single Channel Service, which we define as the smallest executable entity that can run independently. A Single Channel Service is considered the indivisible building block of an application component which can be used to build up more complex services, providing re-usability and extend-ability. As the name indicates, the most important architectural feature of a Single Channel Service is that it communicates over a single communication channel which is basically based on UDP communication. The direct client of a Single Channel Service is the HPA which will act as an interface agent between the service and the web client. A Single Channel Service can be visualized as a Java Servlet which runs in the application server environment and delivers web results to the client. 2- Skeleton Services: Since the Single Channel Service does not differ in concept from a normal web application component, a way is needed to group those independent basic components, the Single Channel Services, to build more complex functionality services able to run those components in parallel to improve performance. A Skeleton Service is basically a server side in-line script which follows the normal structure of regular web server side in-line scripts such as PHP or ASP. Some features are added to the Skeleton to achieve multichannel and parallel execution such as adding parallelization constructs to each in-line code section in the skeleton as well as the type construct defining the development environment of each in-line code section.

The developer will write the skeleton source file which is a hybrid of static content as well as in-line code sections defining the dynamic parts. Then the deployment manager will take as an input the source of the Skeleton to generate the skeleton map and add independent single channel services for each concurrent in-line script section. The

Skeleton map is a map that will be used by the HPA to identify each concurrent service that needs to be requested from the container in parallel. The communication layer of the Container is based on a special state-full protocol built on top of UDP sufficient to serve the web application communication needs of a single request-reply communication sequence. The communication layer consists of multi-threaded components that allow the container to handle multiple communication channels simultaneously and service multiple requests concurrently. The container does not perceive the relation between different channels, rather from the container perspective each communication channel is assigned to a service which either serves a normal service or transfers a skeleton map to the HPA, both of which require a single channel. The HPA is the one which initiates multiple communication channels to different containers to serve a complex service defined by a skeleton map. When a request arrives from the HPA the container starts by validating the client. On successful validation the communication layer passes the HTTP request to the service dispatcher which will then evaluate the HTTP request and with the help of the service factory a communication channel will be assigned to a service to serve the requested web transaction. After the transaction finishes, the communication layer subsystem is responsible for cleaning up the communication channel and re-initializing it to serve future requests. When the HPA initially tries to communicate with a container node, it will do so on a default administrative port through which it will be assigned a range of service ports over which it can request services from the container. The HPA will be able to communicate with any container node in the cluster over the same range of communication ports. The communication layer, with the help of the cluster management subsystem, will assign the HPA to a free range of ports and replicate this assignment to all container nodes in the cluster.

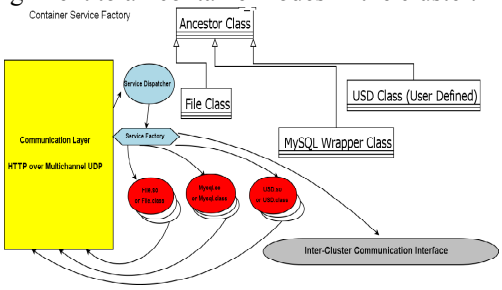


Fig 3. Service Factory

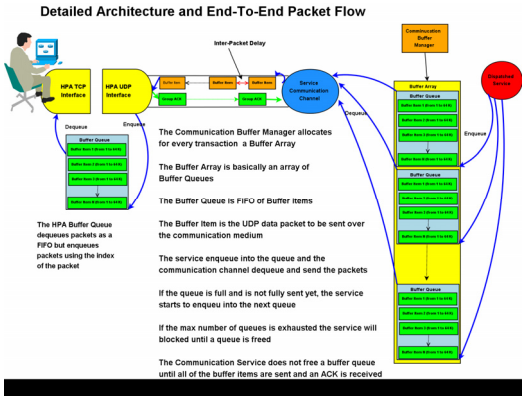


Fig 4. Communication flow between web client and container with HPA in the middle

After a specific idle time from a specific client the port range assignment is cleared and the HPA client will need to reclaim a port range again. The Service Manager subsystem is composed mainly of the Service Manager and the Service Dispatcher which are concerned with the service status in all stages of operations. First a service is loaded by the service factory when it is in the stage of being ready to serve requests. When a request arrives and the service dispatcher decides on the type of service that should serve a specific request, it asks the service factory to avail a service instance for this request, which is the point where the service is assigned by the dispatcher to the communication channel as well as a communication buffer and its status is changed to being operational and dispatched, where it will reside in the active service pool. When the service finishes serving the request, the garbage is collected by the service factory, returned to the status of being ready to use and transferred to the ready to execute service pool. Figure 3 gives an abstract view of the Service Manager and how the Service Dispatcher interacts with the Service Factory. This subsystem is responsible for reading configuration information from configuration sources, which are all based on XML format and require XML parsing, and storing it in internal structures ready for use by different subsystems. For example, the range of ports to be used by the communication layer, the number of instances for a specific service type, . etc. With the help of the Cluster Management System, the Configuration Manager is capable of distributing configuration structures over different container nodes in the web cluster. The Administration Manager is an interface layer between the human administrator of the container web cluster and the container nodes. It enables the administrator to pass administration commands to the container node with the help of the Cluster Management System, the commands issued by the administrator can run transparently on multiple container nodes providing a single system image SSI for the whole container web cluster. The Deployment Manager is responsible for deploying the services provided by the application developers and replicating the deployment over different cluster container nodes with the help of the Cluster Management System.

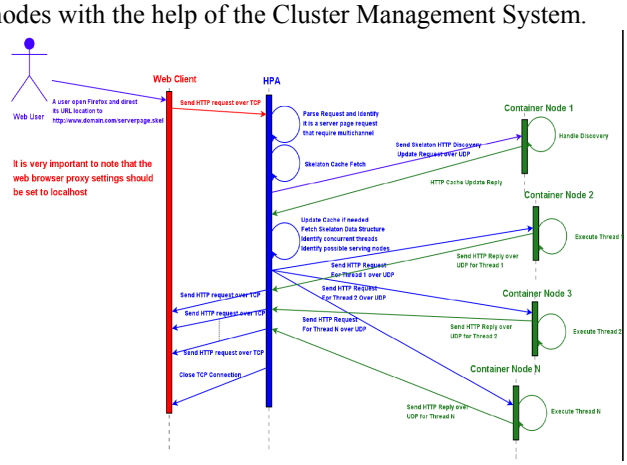


Fig 5. Multi-channel Scenario Work Flow

The deployment manager can deploy single channel services as back-end components as well as multichannel services represented in server side in-line scripts. The developer will provide the multichannel in-line scripts. The

deployment manager will then parse the script and extract each piece of code defined as a separate thread and generate the single channel service source code for it. The deployment manager will then compile the services, generate whatever error or warning messages apply and send them to the deployment administrator. The deployment manager will choose the correct compiler for each code section according to its type, meaning that sections written in C++ will be compiled with GCC for example, and sections written in JAVA will be compiled with an appropriate JAVA compiler. On successful compilation of the services constructed from the in-line script definitions, the deployment agent will deploy those services across the container cluster nodes according to their types. C++ single channel services will be replicated over C++ containers, and JAVA services will be replicated over JAVA containers. It is important to state that some replication constructs and rules can be applied for the service replications. The default replication may be equal distribution of the services, but there might be another deployment scheme which takes into consideration the amount of memory and the speed of the CPU of each container node. After the single channel services are compiled and deployed successfully, the deployment manager will generate a skeleton map for the in-line script and replicate it over cluster nodes. The skeleton map will contain pointers to the target single channel services indicating their primary and secondary locations in case of failures. The service pointer is compose of an HTTP-like header of the request for the single channel service with a little room for adding extra information about the service such as alternative service locations. The Cluster Management System is the subsystem that is responsible for the exchange of information between different containers. The cluster management system enables the deployment manager to distribute newly deployed services as well as modified ones. The Cluster Management System is also responsible for transparently executing administration commands issued by the environment administration over all the nodes of the cluster which eases the administration of the web cluster and makes it appear as a single system to the administrator. Moreover, the Cluster Management Subsystem is responsible for all the communication necessary to carry out the service state migration. The High Performance Agent is the agent that the whole system depends on. The HPA acts as a multi-protocol reverse proxy between the Container and the web client. The HPA acts as a web server for the web client and as the agent which understands the constructs sent by the container to split the communication stream into multiple channels, which will enable the parallelization of delays from which should come the enhanced performance. How the gears will work can be seen in the work flow section. The communication layer of the HPA is a multi-protocol double edged communication layer. It can be viewed as two separate communication layers that communicate with each other. The first communication layer is a standard multi-threaded TCP communication layer that can handle multiple web transactions concurrently. The second, UDP based, communication layer is responsible for communicating with the back end containers. A request is initiated by a web client through an HTTP over TCP connection. When the request arrives to the HPA, the HPA will use one of the already established UDP connections with the container environment and a discovery request will be initiated to identify the node that this request will be served from. A cache for discovery results in the HPA will be updated to eliminate unnecessary communication. Finally, the request will be served from the container to the HPA over UDP and the data stream will be transferred to the web client consequently over TCP, which will take place transparently to the web client. Both communication threads, TCP and UDP, will run in two different threads to avoid dependent communication blocking, hence a buffering mechanism will be needed between the two threads to enable data storage which will help to postpone mutual communication blocking between the two communication threads. Of course when the communication buffer is fully occupied, the UDP receiver will wait until the TCP sender starts spooling data from the buffer and vise versa. Figure 4 gives an overall view of the communication mechanism between the web client and the container with the intermediate agent HPA in the middle. Obviously a server side in-line script will contain some static content, and every time a server side script is requested by the client, the skeleton map for that script will have to be fetched from the container for the HPA to continue and establish the required single channel requests to fulfill serving the server side script request. The connection required to fetch the skeleton map is an overhead, hence adding a cache module to the HPA to keep unmodified versions of skeleton maps will achieve two things: 1) eliminate an extra connection that is needed for the skeleton fetching, 2) cache some of the static content that is embedded in the dynamic content generated by back end services. All the scripted sections will be cached by the HPA, and the impact of that will depend on the size of the cachable areas. Of course in current modern scripting environment such caching is not possible as the client has no clue which parts of the UI, e.g. HTML, is static and which part is generated by a backend business logic engine, yet the client, HPA, in our case has no access to the business logic source code. The Discovery client is the module that is responsible for advising the HPA of the locations of services through communication with the Container discovery service. Caching will be applied to eliminate unneeded communication as much as possible.

II. SYSTEM OPERATION

The work flow of main types of requests and mechanisms is discussed through a file spooler that is used as an example to clarify the three scenarios presented; the Single Channel scenario, the Multi-Channel scenario, and the Service State Migration Scenario [4][5]. Work flow figures provide visualization of each scenario. The Single Channel Scenario is the basic building block upon which the multichannel scenario is built. A special case one container of Figure 5 illustrates the work flow of the single channel scenario. The scenario starts with a web client using the HPA installed on the same machine and operating on the loopback address, to initiate a single channel request to a container node. The request is in normal URI structure which contains the name of the container node that the requested service resides on, and the name of the service to

be executed. The request is sent to the HPA over TCP. The HPA evaluates the request and identifies it as a single channel request. The HPA then opens a UDP connection to the container node specified in the URI, and passes the request to it. The container then dispatches the request to the correct service instance to serve the request. The stream returned by the service to the HPA over UDP is sent to the client over UDP. As can be seen from the figure, the UDP communication is carried out in parallel with the TCP communication which allows the pipelining communication mechanism that eliminates overhead and increases the speed. The multichannel scenario is based on the single channel scenario, as a web transaction is broken down into a number of single channel services that are distributed and executed concurrently and serve their content over parallel communication channels. Figure 5 illustrates the work flow of the multichannel scenario.

The request reaches the HPA over TCP as usual, exactly as in the previous scenario. The HPA evaluates the request and identifies it as a multichannel request by the service name extension .skel. The HPA then makes necessary updates to its skeleton cache. Then it fetches the skeleton data structure from its cache, and identifies the different single channel requests needed. The HPA then spawns a thread for each single channel request to different container nodes according to the information in the skeleton map of the multichannel service. The HPA returns the replies of the channels to the web client over TCP as they arrive according to their chronological order which entails some buffering and blocking techniques. For example, if the second channel finishes before the first channel, the second channel content must be buffered on the HPA side until the first channel finishes, during which the communication channel will be blocked.

III. Configuration Manager (CM)

The CM is basically an XML parser that parses XML configuration files, and an extended data structure that is designed to store all possible configuration constructs stored in configuration files provided by the system administrator. The CM is part of both the Container and the HPA as they both need some initial configuration to be able to start.

In the case of the Container, identical copies of all the configuration files should be present on all Container nodes of the cluster. As we have developed Container nodes running on UNIX environments, the location where such configuration files should be located is "/etc/container/". Three main configuration files should exist for the container node to be able to initialize, start up, and services requests, which are: main.xml, node.xml, service.xml

The main.xml configuration file is an xml file that is responsible for holding all the main configuration necessary for all container nodes to be able to start up. Also it contains some needed configuration that is used by deployment managers that can run from any container node. The main.xml, unlike the other 2 configuration files used by the container, contains only one XML record, as the configuration parameters defined in it are not repetitive. A main.xml file would look like the one below.

```
<MainConfiguration>
<ServiceConfiguration>/etc/container/service.xml</ServiceConfiguration>
<SkelatonPath>/var/www/container/skelaton/</SkelatonPath>
<ServicePath>/var/www/container/services/</ServicePath>
<NodeConfiguration>/etc/container/node.xml</NodeConfiguration>
<StartPort>9000</StartPort>
<JStartPort>10000</JStartPort>
<PortRange>20</PortRange>
<ReserverPort>50000</ReserverPort>
<JReserverPort>60000</JReserverPort>
<ClusterManagementPort>3333</ClusterManagementPort>
<HPAClusterManagementPort>4444</HPAClusterManagementPort>
<JClusterManagementPort>3335</JClusterManagementPort>
<JHPAClusterManagementPort>4445</JHPAClusterManagementPort>
<DeploymentPort>51000</DeploymentPort>
<JDeploymentPort>61000</JDeploymentPort>
<LogFile>/var/www/container/logs/access.log</LogFile>
<LogLevel>5</LogLevel>
<ClientChannels>5</ClientChannels>
<DeployerTempPath>/var/www/container/temp</DeployerTempPath>
<DeployerShell>scp</DeployerShell>
<ContainerObjectPath>/work/Masters/Container/Kdevelop/container/debug/src/</ContainerObjectPath>
<GCCCompiler>/usr/bin/g++</GCCCompiler>
<GCCLinker>/usr/bin/ld</GCCLinker>
<GCCIncludePath>/work/Masters/Container/Common/</GCCIncludePath>
<GCCLibPath>/work/Masters/Container/Common/</GCCLibPath>
<JAVACompiler>/usr/lib/jvm/java-1.5.0-gcj-4.3-1.5.0.0/bin/javac</JAVACompiler>
<JAVAClasspath>/var/www/container/jservices/</JAVAClasspath>
<JAVACClasspath>/work/Masters/Container/Eclipse/Container/bin:/var/www/container/jservices/</JAVACClasspath>
<JServicePath>/var/www/container/jservices/</JServicePath>
</MainConfiguration>
```

As we can see, each container type have different values for some attributes, such as the "StartPort" and the "JStartPort", which allow different container types to be hosted physically on the same hardware node. Of course, if we have more Container types, this configuration files will need to be extended to accommodate that, and consequently the CM will need to be altered to allow for the new attributes to take effect.

The node.xml configuration file is a multi-record XML configuration file. Each XML record in this file represent one node in the cluster and set some of the needed attributes for each node that is necessary for other nodes to be able to communicate with. So a Container node will not need its record in this file as it will not communicate with itself, rather it will need to read all the other records to be able to know how to communicate with other nodes in the cluster. Also, it is important for all nodes in the cluster to know the managed nodes in the cluster to be able to communicate with for cluster specific functionalities. A node.xml file would look like the one below. The XML records in this file is sent to an HPA during the reservation process as a reply for its discovery request to know the needed information about the cluster

```
<Node>
<Hostname>localhost</Hostname>
<IP>10.0.0.172</IP>
<Type>C++</Type>
<ReserverPort>50000</ReserverPort>
<ManagementPort>4444</ManagementPort>
<Role>Management</Role>
</Node>
<Node>
<Hostname>localhost</Hostname>
<IP>10.0.0.172</IP>
<Type>JAVA</Type>
<ReserverPort>60000</ReserverPort>
<ManagementPort>4445</ManagementPort>
<Role>Compute</Role>
</Node>
```

The service.xml configuration file is a multi-record XML configuration file. Each XML record in this file represents a service that the container should load at startup time. Each service have a type which indicates which container type should load. If extra services are loaded by the container as a result of a deployment process, the CM can then update this XML file with its internal buffers with the updates to make these changes permanent, and this will make the container load such services on the next startup. The reason that such configuration file contains services with different types is that more than one container type can be running physically on the same hardware node [6][7][8]. A service.xml file would look like the one below.

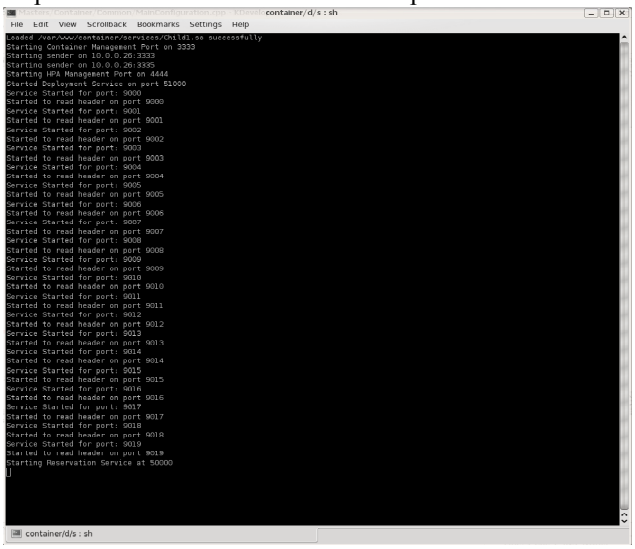
```
<Service>
<Name>Child1</Name>
<Source>com.container.services.TestService</Source>
<Instances>10</Instances>
<URL>/TestService.java</URL>
<Type>JAVA</Type>
</Service>
<Service>
<Name>Child1</Name>
<Source>Child1.so</Source>
<Instances>100</Instances>
<URL>/CHILD1.c++</URL>
<Type>C++</Type>
</Service>
```

The HPA has only one configuration file that direct it to the Managment Node of the cluster from which it will receive extra data about the cluster. The configuration file is called also nodes.xml and it contains multiple XML records; each record represents the necessary details of a management node together with its rank or importance. An HPA node.xml file would look like the one below.

```
<ManagementNode>
<IP>10.0.0.172</IP>
<ManagementPort>4444</ManagementPort>
<Rank>1</Rank>
</ManagementNode>
<ManagementNode>
<IP>10.0.0.173</IP>
<ManagementPort>4444</ManagementPort>
<Rank>2</Rank>
</ManagementNode>
```

IV. EXPERIMENTS

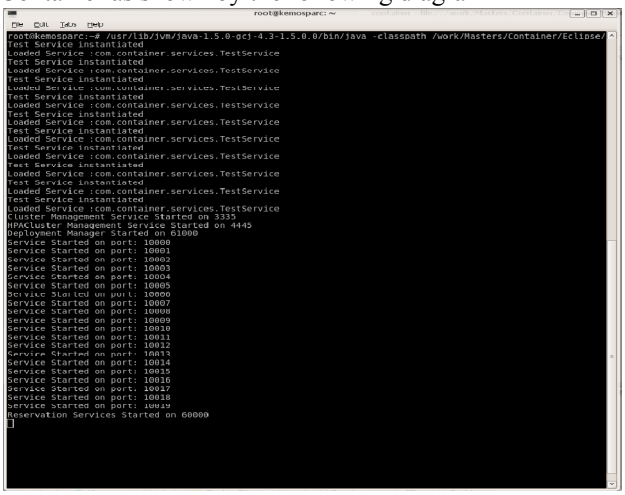
We firstly show snapshots of the container and the HPA while they are starting up, and how the configuration stored in the configuration files and loaded by the CM reacts on the messages printed on the console of each. The following snapshot shows a C++ container startup console.



As can be seen the container performs the following steps during its startup:

1. The container starts by loading the services defined in the service.xml
2. The container then starts its management service on the port defined in main.xml
3. The container then starts management client senders to all nodes in the cluster.
4. The container then starts the HPA management service on the port defined in main.xml, if it is a management node as defined in node.xml
5. The container then starts the deployment service on the port defined in main.xml
6. The container then starts its channels on the range of ports defined in main.xml depending on its type.
7. Finally, the container starts the reserve service in the port defined in main.xml depending on its type.

The same sequence of steps are performed by a Java Container as shown by the following diagram



The HPA is performing the following steps during its startup:

1. The HPA gets the management node address and port from its configuration
2. The HPA then sends a discovery request to the management node to get addresses and reservation ports of available nodes in the cluster.
3. The HPA starts the channel reservation process iteratively with each node in the discovery list returned.
4. Finally the HPA starts the HPA Client Management service and connects to all management nodes in the cluster.

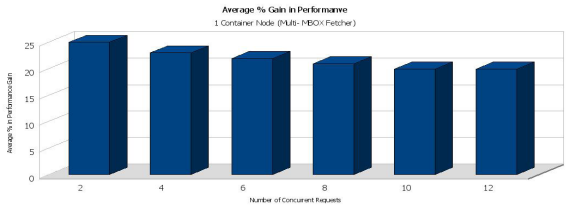


Fig 6: 1 Server Node (Multi-Mailbox Email Web Console)

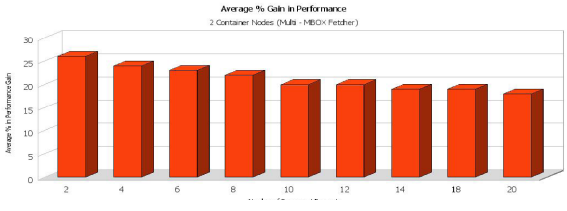


Fig 7: 2 Server Nodes (Multi-Mailbox Email Web Console)

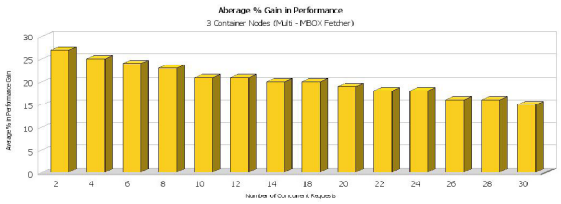


Fig 8: 3 Server Nodes (Multi-Mailbox Email Web Console)

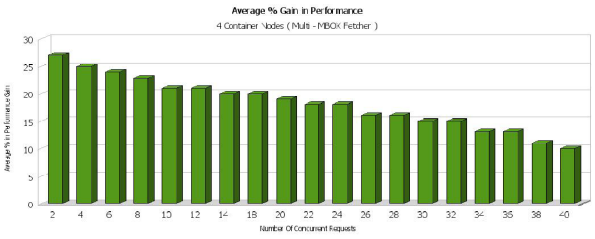


Fig 9: 4 Server Nodes (Multi-Mailbox Email Web Console)

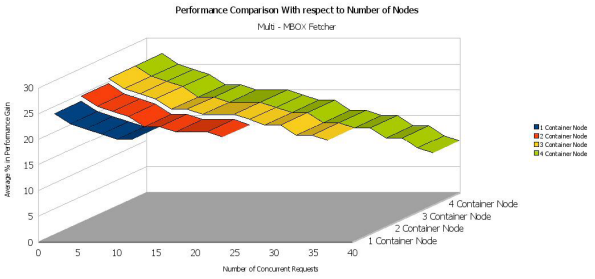


Fig 10: Performance Comparison with respect to Number of Nodes (Multi-Mailbox Email Web Console)

V. Case Studies: Multi Account Email Web Console And RSS Aggregator

Nowadays, access to email is very important for everyone, and not all private email environments have web email interfaces, so POP email services are available on the web to pop a users email through the POP protocol and make the user's mailbox accessible from the web browser, which increases the accessibility of the email from any location eliminating the need for an email client software such as Thunderbird or Evolution. Moreover, a user may have multiple email boxes at physically different email servers that he/she might want to access through a single console. Such cases would benefit very much from the multichannel environment which would allow fetching emails from different mailboxes in parallel, while on the contrary, a traditional web application will pop the users email from his mail boxes one after the other. This kind of application is very interesting, although as in the first case study, some delay and processing will be needed to fetch the email, yet what differs from the first case study is that the back-end application will use and share a single INTERNET link to access the mail box which will act as a bottleneck when the link is fully utilized; hence the gain in performance will be limited due to the wait time that the concurrent threads will be subject to as a result of smaller shared connection slices.

The runs of the experiments were carried out on both environments, the traditional and the multichannel. Two variables were changed throughout the experiment runs, which were the number of concurrent requests, and the number of nodes in the serving cluster. The same experiments and steps were followed as in the previous application. Figures 6-10 illustrate the results of the experiments.

Form the above results, the following are the most important observations:

1. The multichannel environment did not provided a high performance gain as in the previous case study. The average percentage of performance gain was around 19%.
2. The gain in performance was directly proportional to the increase in the number of nodes in the cluster, and depreciated slightly as the number of concurrent connections increased.
3. The execution duration of the request was shorter with respect to the web usage statistics case study.
4. The clustering and adding new nodes in this case study did not have a strong impact on the performance gain, which was very clear from the last graph.
5. It was noticed during the experiments that the processing usage during the multichannel experiments was slightly higher than the processing usage during the traditional environment experiments.

The second case study is the Web based RSS Aggregator. An RSS collector is basically a web front-end that consolidate more than one RSS feed from different sources. The application is basically an on-line web-based RSS aggregator that initiates curl-like calls to different RSS sources identified from the profile of a specific user. The application is fairly simple, and requires less processing power than web usage statistics gathering or email aggregation, yet some minor delays at the back-end is often observed due to communication with the different RSS servers. The runs of the experiments were carried out on environments, the traditional and the multichannel. Two variables were varied throughout the experiment runs, the number of concurrent requests, and the number of nodes in the serving cluster.

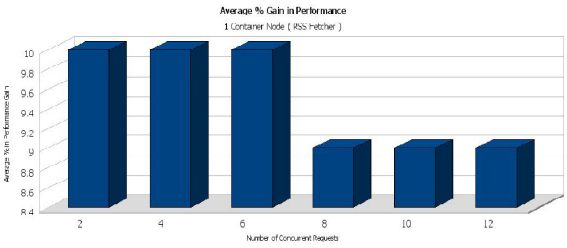


Fig 11: 1 Server Node (Web-based RSS Aggregator)

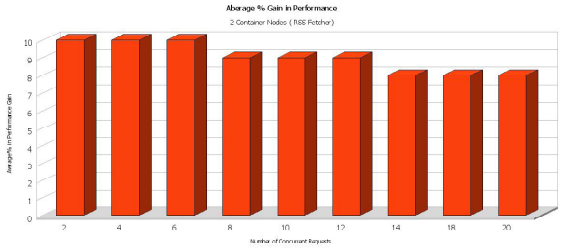


Fig 12: 2 Server Nodes (Web-based RSS Aggregator)

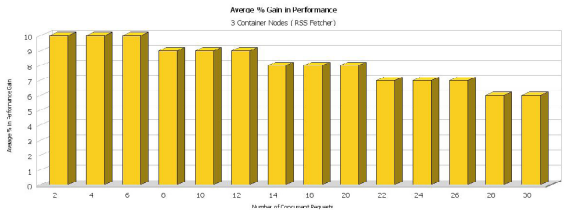


Fig 13: 3 Server Nodes (Web-based RSS Aggregator)

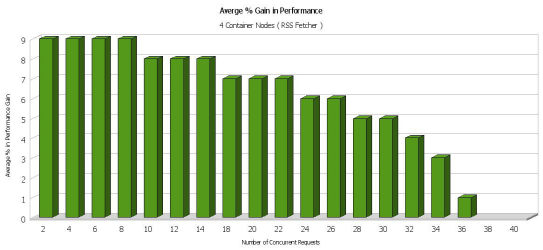


Fig 14: 4 Server Nodes (Web-based RSS Aggregator)

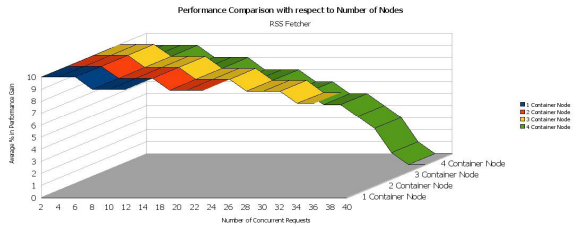


Fig 15: Performance Comparison with respect to number of nodes (Web-based RSS Aggregator)

The same experiments and steps were followed as for the previous applications. Figures 11-15 illustrate the results. The following are the observations

1. In this case study the improvement in performance due to the multichannel environment was minimal, and the average percentage of performance gain is around 7%.

2. The execution duration of the request was very short compared to the two previous case studies.

3. The clustering and adding of new nodes had minimal effect on the performance gain at the beginning and as the number of nodes increased as well as the number of concurrent requests. The clustering had zero effect on the performance.

4. It was noticed during the experiments that processing usage during the multichannel experiments was almost the same as the processing usage during the traditional environment experiments.

Our analysis for the above three case studies targets two main issues: why the performance gain degraded from the first case study to the third one, and what are the best characteristics of the applications that will most benefit from multichannel environments. Tackling the first point, it is very obvious that the Web Usage Statistics application made maximum benefit from the multi-channel environment. A deeper look into the characteristics of such applications reveals that this was due to two main reasons:

1. The duration of the back-end execution of the service required to generate the web usage statistics is relatively long, and requires much processing on the back-end, which provides many process delay parallelization chances that the multichannel environment can utilize unlike the traditional web application environment. In the other two case studies, the Multi Account Email Web Console and the Web based

RSS Aggregator, the request fulfillment duration is relatively small and is based on data transfer rather than processing leaving few opportunities for improvements in processing speed to improve performance, and leading to a degradation in the rate of gain in performance.

2. The Web Usage Statistics Application is based on external service, as are all three applications in the three case studies. The external service in the case of the web usage statistics is a database service, and in this case the service can be expanded to run on different nodes with database replication running in the background which allows data consistency and independent processing resources for every execution channel. The other two applications, the INTERNET connection through which the applications communicate with the external services they depend on, namely the RSS web server for the RSS Aggregator and email servers for the Multi Account Email Web Console. The means of communication, the INTERNET connection, that both applications depend on, and their requests utilize in a sharing model, presents a bottle-neck that prevented the multi-channel environment from utilizing the inherent concurrency in the applications. It is very obvious that all three chosen applications are inherently parallel; the Web Usage Statistics can run usage queries over a usage database distributed and in parallel independently, the Multi Account Email Console can fetch emails from different mail boxes independently and in parallel, and the RSS Aggregator can fetch RSS feeds from different sources in parallel as well, but the most important issues that affected the performance were the deployment architecture and the nature of the applications. It is very important to highlight that both of the above two points affected the performance results of the experiments dramatically, which can be shown by the three charts below that show the degradation in the rate of increase of the performance with respect to the increase in the number of concurrent requests. It is obvious from the above graph on adding the new client machine the gain in performance in the fifth returned back to the normal increase compared to the previous runs. Figures 16-20 compare the three applications in the three case studies with respect to their gain in performance related to clustering and show the domination of the Web Usage Statistics application over the other two applications. Tackling the second target of the analysis, web applications that will benefit most from the multi-channel environment should posses three main characteristics:

1. The application should be inherently parallel, meaning that it can be broken into smaller independent threads of execution, which is a very common characteristic in web applications in general. Moreover, when the application fragments are more closely equal in size (Execution Duration) better results can be achieved.

2. The amount of execution needed by the application should be relatively big with respect to the amount of I/O.

3. The threads of execution within a request, which are represented by web channels, should depend, as little as possible, on limited shared resources that waste the parallelism, and force running threads to stand waiting in queues which will emulate sequential processing and defeat the whole purpose.

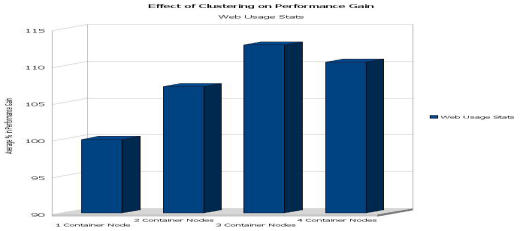


Fig 16: Effect of Clustering on Performance Gain (Web Usage)

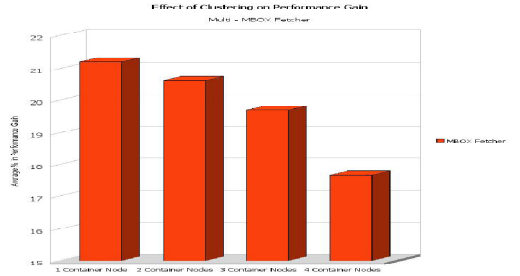


Fig 17: Effect of Clustering on Performance Gain (Multi-Mailbox Email Web Console)

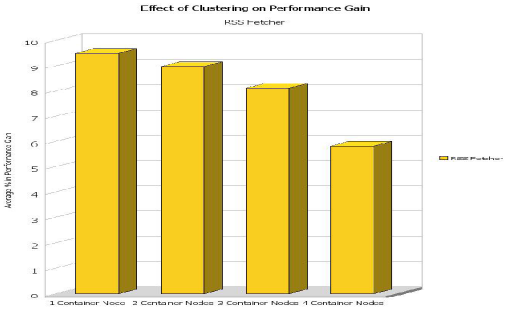


Fig 18: Effect of Clustering on Performance Gain (RSS Aggregator)

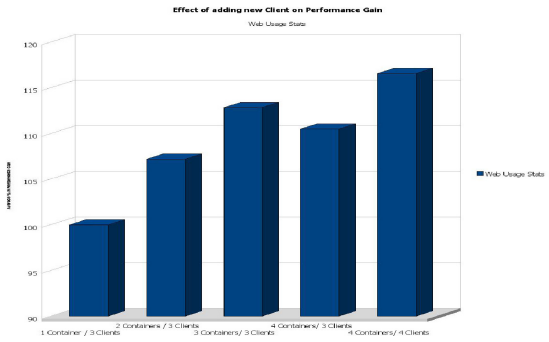


Fig 19: Effect of Clustering on Performance Gain (RSS Aggregator)

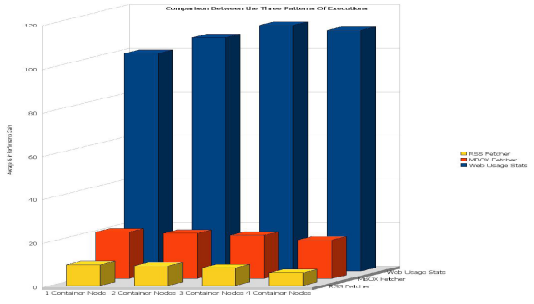


Fig 20: Comparison between the three Patterns of Execution

VI. Conclusion and Future Work

This research is an attempt to find an alternative web architecture that can provide better performance for processing intensive web applications. As the technology advances and becomes more complex, the window for coming up with new inventions narrows relatively, yet what can be done is to use what already exists from technology, standards, invention, and engineering techniques to restructure what is available and borrow mechanisms and ideas from different fields of computer science, and research their validity and their beneficial impact in other domains of computer science. The approach followed in this research is to invite ideas, techniques, standards, architectures, and models of processing from different domains in computer science to the web environment. The claim was that the technologies and architecture of current web environments are not adequate to serve all patterns of execution, especially processing intensive web applications. Due to the numerous outstanding characteristics of the current web environment such as its standards, extendibility, expandability, and openness an architectural change is needed to make a new generation or strain of web environments available to serve such needs. The key challenge in this research is that the current web environment is a very well established environment and its standard, especially the interfacing standards are very widespread and depended on by all web clients, thus major changes in the architecture should be made in a way that achieves the needed target without changing the client interfaces. As the target from the beginning was performance, HPC environments and conventions were a very appealing and diversified area from which many ideas were borrowed. Through the restructuring of the web environment, by changing the internal communication layer and basing our solution on clustering, a multichannel environment was attained that not only provides better performance in a specific domain of applications, but the architectural change also contributed to extending already existing web environment characteristics and making more use of already existing features such as the scripting model for web programming, business logic isolation, and web caching. Other new features were also introduced, which were born from the architectural changes applied, such as the service state migration and the fault tolerance takeover mechanism. It is very important though to stress that this paper did not target by any means the invention of new network protocols, dispatching and load balancing algorithms, fault tolerance solutions, or automatic application parallelization, rather it provided an architectural framework that will provide a base for application developers to utilize to achieve all of the above. The targets achieved in this research can be summarized as follows:

1. Multichannel environment provides better performance for applications that require considerable need for processing with respect to I/O.
2. Multichannel environment makes more use of clustering than traditional environment, due to its ability to fragment web server pages scripts.
3. A complementary web environment is achieved, which can co-exist side by side with traditional environments, instead of being an alternative environment.
4. Borrowing ideas, mechanisms, and implementation techniques from other fields of computer science such as HPC proved to be very beneficial.
5. Web client interface is kept unchanged, and the changes in architecture were made transparently.
6. Service state migration can occur between two containers of different development technologies.
7. Fault tolerance, through service state migration, recovers a failing request without the need to re-execute the failing request.
8. Different services and service portions can exchange messages through replicated shared memory segment.
9. A single server page can be written in more than one development technology.
10. A multichannel clustered environment can be composed of different containers with different development technologies.
11. Deployment is less complex and does not need a lot of configuration.
12. Smarter caching mechanisms were achieved through skeleton map caching allowing the caching of static content existing within dynamic scripts.

Hopefully, a web environment has been reached that provides better performance for a specific domain of web

applications that are inherently parallel, and need a considerable amount of processing power compared to I/O. Through the case studies chapter we tried to show the most important characteristics of applications which will benefit from the multichannel environment. It is very important to highlight that fusing ideas coming from different domains needs a lot of prototyping and testing, as ideas that appear matching and coherent might require a lot of workarounds and tweaking to achieve integration in reality, thus a considerable amount of the effort expended in this paper was directed to detailed technical design, prototyping, coding, and testing. Finally, this paper proposed a new environment through some new and non-traditional architectural changes, and researched the effect of such architectural changes on performance as well as other aspects of web environments.

REFERENCES

[1] Ahmed Sameh, Karim Sobh, "A Clustered Web Application Server - Architecture", International Journal of Advanced Engineering Sciences, Vloume 4, Issue 1, March 2011.

[2] Ahmed Sameh, Karim Sobh, "Multi-Channel Clustered Web Application Servers - Deployment Manager", to appear in the Proceedings of the IEEE conference on Cloud Computing & Visualization- CCV 2011, Malysia, April 25-26, 2011.

[3] Ahmed Sameh, Karim Sobh, "Multi-Channel Clustered Web Application Servers - Cluster Manager", Working paper.

[4] Jon Crowcroft, Iain Phillips. TCP/IP and Linux Protocol Implementation: Systems Code for the Linux Internet. John Wiley Sons, Dec 2001. ISBN-10: 0471408824. ISBN-13: 978-0471408826.

[5] UDP-based Data Transfer (UDT) http://udt.sourceforge.net/.

[6] Gigabit Ethernet Jumbo Frames http://sd.wareonearth.com/phil/jumbo.html.

[7] Path MTU Discovery - RFC 1191http://www.faqs.org/rfcs/rfc1191.html.

[8] MTU Limits http://www.psc.edu/mathis/MTU/limits.html.

Ahmed Sameh, Correspondence Author Alexandria, Egypt in 1957, Moved to Canada in 1980. He is now dual citizen. He earned his B.Sc., M.Sc., and Ph.D. degrees from both Alexandria University and the University of Alberta in 1979, 1984, and 1989 respectively. All degrees are in Computer Science and Engineering. Prof. Sameh is currently a Professor of Computer Science at Prince Sultan University, Riyadh, Saudi Arabia. Before that he was with the American University in Cairo. He did hold positions at the George Washington University, University of Kuwait, and Lewis & Clark at various stages in his academic life. His major areas of research are in High Performance Computing, Artificial Intelligence, Neural Networks, Mobile Computing and Wireless Communications. He has several publications including ten book chapters, fifty archival journal papers, and one hundred and seventy refereed conference papers. These publications are mainly in the areas of Cluster and Grid Computing, Mobile Computing, and Neural Networks. Prof. Sameh is a member of the IEEE, ACM, ECS, CIPS, and ISCA. He has received many honors and awards. He has participated in many conference organizations, and journal/conference refereeing.

Karim Sobh is the owner and CEO of "Code-Corner"- Solution-Oriented IT consultant. He holds a B.Sc. degree in Computer Science with honors from the American University in Cairo, 1997. His experience falls within Cluster Management, Open Source development, System Programming, and Internet Security. Karim holds several international Professional certifications from Jedwards, Tivoli Academy, and IBM. He is currently a graduate student in the department of Computer Science and Engineering at the American University in Cairo. He has a good record of scientific publications.