Decentralized Orchestration of Composite Web Services

Girish Chafle
IBM, India Research Laboratory
New Delhi, India
cgirish@in.ibm.com

Sunil Chandra
IBM, India Research Laboratory
New Delhi, India
csunil@in.ibm.com

Vijay Mann
IBM, India Research Laboratory
New Delhi, India
vijamann@in.ibm.com

Mangala Gowri Nanda
IBM, India Research Laboratory
New Delhi, India
mgowri@in.ibm.com

ABSTRACT

Web services make information and software available programmatically via the Internet and may be used as building blocks for applications. A composite web service is one that is built using multiple component web services and is typically specified using a language such as BPEL4WS or WSIPL. Once its specification has been developed, the composite service may be *orchestrated* either in a *centralized* or in a *decentralized* fashion. Decentralized orchestration offers performance improvements in terms of increased throughput and scalability and lower response time. However, decentralized orchestration also brings additional complexity to the system in terms of error recovery and fault handling. Further, incorrect design of a decentralized system can lead to potential deadlock or non-optimal usage of system resources. This paper investigates build time and runtime issues related to decentralized orchestration of composite web services. We support our design decisions with performance results obtained on a decentralized setup using BPEL4WS to describe the composite web services and BPWS4J as the underlying runtime environment to orchestrate them.

Categories and Subject Descriptors

H.3.5 [**Information Storage and Retrieval**]: Online Information Services—*Web-based services*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Distributed Programming, Parallel Programming*

General Terms

Design, Performance, Experimentation

Keywords

Composite Web Services, Decentralized Orchestration, Code Partitioning, BPEL4WS

1. INTRODUCTION

Web services encapsulate information, software or other resources, and make them available over the network via standard interfaces and protocols [10]. Complex web services may be created by aggregating the functionality provided by simpler ones. This is referred to as *service composition* and the aggregated web service becomes a *composite web service*.

Composite web services may be developed using a specification language such as BPEL4WS [2, 13], WSIPL [8], WSCI [5], *etc* and executed by an engine such as BPWS4J [1]. Typically, a composite web service specification is executed by a single coordinator node. It receives the client requests, makes the required data transformations and invokes the component web services as per the specification. We refer to this mode of execution as *centralized* orchestration. The coordinator node is responsible for coordination of all data and control flow between the components, and hence becomes a performance bottleneck. All data is transferred between the various components via the coordinator node instead of being transferred directly from the point of generation to the point of consumption. This leads to unnecessary traffic on the network. In addition, it is possible that a web service generates a lot of data that is irrelevant to the composite service, yet this data will be transferred to the coordinator node where it is discarded, thereby putting unnecessary load on the network. These factors lead to poor scalability and performance degradation at high loads.

Specifying a composite service using a language like BPEL4WS has interesting ramifications. The specification can be analyzed using techniques such as program analysis [16], petri-nets [21],etc. The data and control dependences between the components can be analyzed and the code can be partitioned into smaller components that execute at distributed locations. We refer to this mode of execution as *decentralized* orchestration. In decentralized orchestration, there are multiple engines, each executing a composite web service specification (a portion of the original composite web service specification but complete in itself) at distributed locations. The engines communicate directly with each other (rather than through a central coordinator) to transfer data and control when necessary in an asynchronous manner. It may appear that the introduction of additional engines in the execution path would adversely affect performance, however decentralized execution brings performance benefits for the following reasons :

- There is no centralized coordinator which can be a potential bottleneck.
- Distributing the data reduces network traffic and improves transfer time.

WWW2004, May 17–22, 2004, New York, New York, USA.
ACM 1-58113-912-8/04/0005.

- Distributing the control improves concurrency.
- Asynchronous messaging between engines brings benefits of better throughput and graceful degradation [11].

Furthermore, decentralized orchestration might be the only way to compose web services in constrained data flow environments (e.g. business-to-business scenarios) where data might flow only in a given direction due to business constraints. This inevitability of decentralized orchestration in constrained data flow environments and the potential performance benefits to be gained through decentralized orchestration motivated us to explore it further.

While decentralization brings performance benefits, it also increases the complexity of the system and poses many build time and runtime challenges. It requires modifications to the infrastructure to execute the service, build time and runtime support for error handling and recovery, and techniques/tools for code partitioning. These complex build time and runtime issues have to be properly addressed in order to exploit the full performance benefits of decentralization.

In this paper we identify various runtime and build time issues in decentralized orchestration of composite web services and discuss solutions to address them. The build time issues discussed in Section 2 include :

- determining how to efficiently partition the centralized specification
- distributing error handling code across partitions so as to correctly and efficiently handle runtime errors

The runtime issues discussed in Section 3 include :

- using an efficient protocol for engine to engine communication
- internal design details of the engine to avoid potential deadlocks
- infrastructure for error handling and recovery

We experimentally evaluate the performance of decentralized orchestration and the effect of various runtime and build time issues on performance. The experiments were conducted using BPEL4WS to describe the composite web service and BPWS4J to orchestrate it. These results are presented in Section 4. The related work is summarized in Section 5 followed by conclusions and future work in Section 6.

2. BUILD TIME ISSUES

This section explains decentralized orchestration by taking a sample composite web service specification and decentralizing it. We also describe the various build time issues that come up during this process. For the purpose of illustration, we use a composite web service – the *FindRoute* service, whose task is to find the driving directions from the address of one person to that of other. It takes as input the names of the two people, gets and collates the necessary information from three component web services and returns the driving directions from the first address to the second address.

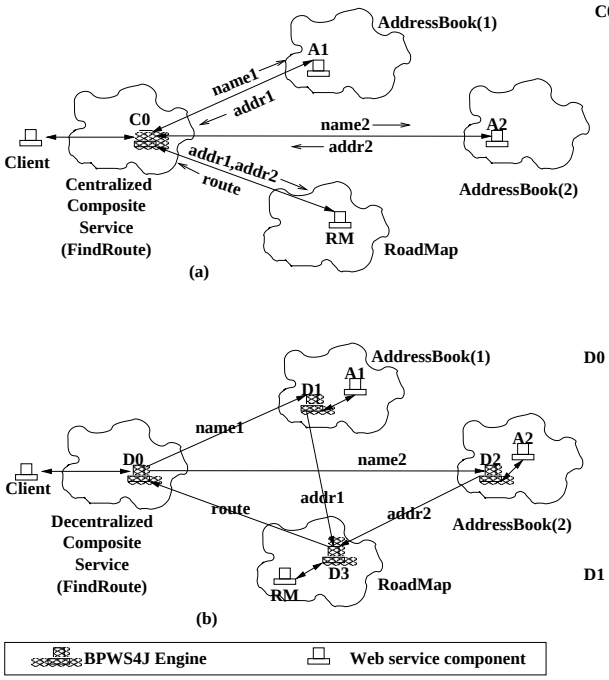
Centralized Orchestration. In centralized orchestration, the *FindRoute* composite service receives two names *name1* and *name2* from a client, then sends *name1* to a web service *AddrBook(1)* which returns the address, *addr1*, of *name1*. Concurrently, *FindRoute* sends *name2* to a web service *AddrBook(2)* which returns the address, *addr2*, of *name2*. From the two addresses returned, *FindRoute* extracts just the *city* and *zip* of the two addresses and sends this to a web service *RoadMap* which computes driving directions from one location to the other. These directions are returned to the client. The service is graphically represented in Figure 1(a) and the corresponding (pseudo) BPEL4WS specification (refer appendix A) is shown in Figure 1(c).

Decentralized Orchestration. In decentralized orchestration of *FindRoute service*, the BPEL4WS code is partitioned into four partitions - `D0`, `D1`, `D2` and `D3`, which are executed by four BPWS-4J engines at the four locations. Figure 1(b) graphically depicts the decentralized orchestration of *FindRoute Service* and the corresponding pseudo-BPEL4WS specification is shown in Figure 1(d). The `D0` partition receives two names *name1* and *name2* from a client, then sends *name1* to partition `D1` and *name2* to partition `D2` asynchronously in parallel and then waits on a callback `receive` for the results from `D3`. The `D1` partition invokes web service *AddrBook(1)* synchronously passing in *name1* as the input and receiving address, *addr1*, of *name1* in return. Similarly, the `D2` partition invokes web service *AddrBook(2)* with *name2* as the input and receives address, *addr2*, of *name2* as the response. Only the relevant *city* and *zip* information is extracted from the address and sent from `D1` to `D3` and similarly from `D2` to `D3`, thus reducing the data on the network. The `D3` partition waits till it receives the inputs from both `D1` and `D2` and then invokes the *RoadMap* web service that returns the driving directions from one address to the other. The `D3` partition returns the driving directions back to `D0` partition through a callback. `D0`, on getting the callback, receives the driving directions and sends them back to the client.

2.1 Code Partitioning

In decentralized orchestration, the various interactions between the components are analyzed and the composite web service specification is partitioned using program analysis techniques. We have built a tool that does this task automatically. The partitions are full-fledged composite web service specifications themselves, that execute at distributed locations (preferably collocated with the web services) and can be invoked remotely. Our tool also generates the *Web Service Description Language* (WSDL) [9] descriptors for each of these fragments. The WSDL descriptors permit them to be deployed and invoked like any standard web service.

The code partitioning algorithm identifies the number of final partitions based on the number of component web services in the composite web service. Then the centralized code specification is partitioned across the components in such a manner that the data being passed between components is minimized and the parallelism amongst the components is maximized. The build time process of decentralization essentially consists of three steps - (1) automatic parallelization and code partitioning, (2) synchronization analysis and (3) code generation. Although BPEL4WS permits specification of explicit parallelism, we still apply data flow analysis [16] techniques to determine the maximum parallelism and then apply a cost function to determine the most efficient code partition - which may or may not make use of all possible parallelism. After the code has been partitioned the interactions between the partitions are analyzed to determine the best synchronization protocols to use between the partitions [17]. Once this analysis is complete, the BPEL4WS code for the partitions is generated - an example is shown in Figure 1 (d). The issues of code partitioning are mentioned here for sake of completeness but we do not address it in this paper. These are discussed in an earlier paper [16].



C0
```
receive(client, c{name1,name2})
flow
   sequence
      n1.name = c.name1
      invoke(A1, n1{name}, a1{ph,street,city,zip})
   end−sequence
   sequence
      n2.name = c.name2
      invoke(A2, n2{name}, a2{ph,street,city,zip})
   end−sequence
end−flow
r.city1 = a1.city; r.city2 = a2.city
r.zip1 = a1.zip; r.zip2 = a2.zip
invoke(RM, r{city1,city2,zip1,zip2}, dir{routes})
reply(client, dir)
```
(c)

D0
```
receive(client, c{name1,name2})
n1.name = c.name1
n2.name = c.name2
flow
   invoke(D1, n1{name})
   invoke(D2, n2{name})
end−flow
receive(D3, dir)
reply(client, dir)
```

D1
```
receive(D0, n1{name})
invoke(A1, n1{name}, a1{ph,street,city,zip})
r1.city = a1.city
r1.zip = a1.zip
invoke(D3, r1{city, zip})
```

D2
```
receive(D0, n2{name})
invoke(A2, n2{name}, a2{ph,street,city,zip})
r2.city = a2.city
r2.zip = a2.zip
invoke(D3, r2{city, zip})
```

D3
```
flow
   receive(D1, r1{city, zip})
   receive(D2, r2{city, zip})
end−flow
r.city1 = r1.city; r.zip1 = r1.zip
r.city2 = r2.city; r.zip2 = r2.zip
invoke(RM, r{city1,city2,zip1,zip2}, dir{routes})
invoke(D0, dir)
```
(d)

Figure 1: Centralized and Decentralized Orchestration

2.2 Handling Errors at Build Time

The use of asynchronous messaging between the different composite web service partitions makes error propagation in a decentralized setup more complex. BPEL4WS provides a mechanism to explicitly catch errors and handle them by executing subroutines specified in fault handler elements. In decentralized orchestration, one of the challenges is to partition the existing fault handlers in the composite web service specification correctly so that they retain their semantics even after partitioning. If the fault handler includes sending a message to some other component in the composite web service (which now executes on a different node due to decentralization), changes have to be made accordingly.

For example, there can be a fault handler in the centralized FindRoute specification in Figure 1(c) which catches all errors arising out of the address book web service invocation failures and returns an error response back to the client without invoking the road route service. It might look something like this:

```
sequence
    n1.name=c.name1
    invoke(A1,n1(name),a1{ph,street,city,zip})
    throw (''addrBookFailure'')
end-sequence
...
...
faultHandlers
    catch (''addrBookFailure'')
    errorMsg.msg = ''Address Book service not available''
    reply(client, errorMsg)
end-faultHandlers
```

While partitioning the centralized specification this fault handler should be moved to the partitions controlling the address book services (partitions `D1` and `D2`). Furthermore, the fault handler should now send an error message to the partition `D3` since it might be waiting for an input from one of the address book services (provided that the other address book service executed successfully). The error message here could be in the form of an invalid input (e.g. value of -1 for the zip code). In this case the fault handler might look something like this:

```
faultHandlers
    catch (''addrBookFailure'')
    r1.zip = -1
    invoke(D3, r1{city, zip})
end-faultHandlers
```

Partition `D3` can check for the erroneous input and send back an error to the first node.

```
flow
    receive(D1,r1{city,zip})
    receive(D2,r2{city, zip})
end-flow
if (r1.zip >0 && r2.zip >0)
    r.city1=r1.city; r.zip1=r1.zip
    r.city2=r2.city; r.zip2=r2.zip
    invoke(RR,r{city1,city2,zip1,zip2},dir{route})
end-if
else
    dir.route = ''Address Book service not available''
end-else
invoke(D0,dir)
```

Another challenge is to insert additional fault handlers so that errors propagate correctly in the decentralized setup. This is not an issue in centralized setup as the control remains centralized and all errors are propagated back to the client gracefully. In case of decentralization, errors are localized at their respective node of occurrence due to the use of asynchronous messaging. This can be alleviated by inserting additional fault handlers in each of the composite service fragments. These fault handlers can then either invoke a centralized entity or an activity waiting for an input from the failed activity informing it about the error. The error message can then be sent back to the client. For example, the fault handler described above, should be inserted in the partitions `D1` and `D2` and the corresponding check should be made in partition `D3`, even when the original centralized web service specification doesn't have any fault handler.

3. RUNTIME ISSUES

Decentralized orchestration involves partitioning a centralized composite service specification into smaller partitions that are full-fledged web services. Each partition requires a runtime environment (a workflow engine) for execution. Thus decentralized orchestration introduces multiple engines communicating with each other. This communication appears as a web service invocation in each partition (typically as the last step of the flow). Different parameters affect the efficiency of this communication - the protocol used, the threading models used in the engine, etc. Furthermore, a runtime infrastructure is also needed for error propagation and error recovery in decentralized orchestration which is complicated due the use of asynchronous messaging. We discuss these issues in detail in the following subsections.

3.1 Application Server and Messaging

Composite web services are executed by an engine such as BPWS4J. The engine itself is hosted as a web service inside a web service container. The implementation of a web service container depends on the type of protocol it uses to communicate with its clients. SOAP-over-HTTP and SOAP-over-JMS are the common messaging protocols used for invoking web services. Therefore, we will restrict our discussion to these two protocols.

3.1.1 HTTP based Application Server

HTTP is a synchronous protocol, whereas decentralized orchestration requires asynchronous messaging. "Pseudo" one-way messaging can be achieved over HTTP by sending a *dummy* response to the client. When a web service is invoked using HTTP as the messaging protocol, its hosting web service container is typically implemented as a servlet, hosted inside a servlet container. As servlets were designed for synchronous request-response invocations, a limitation of the servlet container is that a response cannot be sent back to the client unless the servlet thread completes the processing of the message. Thus, a servlet cannot accept a request in a thread, then close the connection and continue to process the request asynchronously in the same thread. As a result, a client is blocked as long as the request is being processed at the server. The BPWS4J engine solves this problem by maintaining an *internal* thread pool (which is independent of the servlet thread pool). When the engine receives an asynchronous request, it puts the request into an internal queue, creates a *dummy* response and sends it back via the servlet thread. The servlet thread is then free to receive a new request and the client is also not blocked. Meanwhile, the asynchronous request is processed in one of the threads from the *internal* thread pool. Both the thread pools are of comparable size.

3.1.2 JMS based Application Server

When a web service is invoked using JMS as the messaging protocol, the hosting web service container is typically implemented as a Message Driven Bean (MDB) [3]. This MDB is hosted inside an Enterprise Java Bean(EJB) container. The EJB container has a single thread (listener thread) listening on a specified topic/queue. It also has a thread pool to process the incoming requests (processing thread pool). On receiving a request, the listener thread puts it on an internal queue and continues listening for more incoming requests. One of the threads from the processing thread pool picks up this request and processes it. If the request is synchronous, the processing thread, after completing its processing, extracts the client address from the context and sends the reply. If the request is asynchronous, nothing is sent back to the client.

3.1.3 Performance Comparison

At a first glance, there is not much difference between an asynchronous message handled by an EJB container or a servlet container. In both the cases a listener thread receives a request, puts the request into a queue and continues to listen for more requests while the request is processed in a thread in the background. In fact, at low loads the performance of a decentralized orchestration over HTTP is similar to a decentralized orchestration over JMS. However, at high loads, experimental analysis (details in Section 4) shows that decentralized execution over HTTP fares poorly compared to a JMS solution. The reason is as follows: When an engine `D1` sends an asynchronous HTTP message to another engine `D3`, `D3` parses the request, determines that it is asynchronous, puts it into a queue and immediately returns a dummy response to `D1`. At low loads this process takes less than 30 milliseconds on an average. At high loads, when the system is loaded and many threads are executing concurrently, it has been observed that the thread at `D3` gets descheduled before it can send the response and so the turn around time can be as high as 2000 to 3000 milliseconds. The consequence of this is two-fold. Both the processing thread at the sending engine and the receiver thread at the receiving engine are blocked for longer periods of time. This reduces their capacity to process other requests and hence the throughput reduces and the response time increases. Asynchronous web service invocation over JMS does not suffer from this limitation as it does not have to send back a dummy message.

3.2 A Potential Deadlock

As a result of decentralized orchestration, a potential deadlock situation might arise under the following conditions:

- The engine maintains a processing thread pool i.e. it does not spawn a new thread whenever there is a new request, and
- Composite web service instances have thread affinity.

Consider a BPEL4WS program that has a correlated `receive` (a receive waiting for an event with correlation information that is used for locating an existing composite service instance) as in `D0` in Figure 1(d). The first `receive` instantiates a BPWS4J composite service instance in a thread. This instance cannot exit until it gets a message that satisfies the second `receive` and hence blocks a thread until the second message is received. When the second message arrives, it is picked up by another thread from the *same* thread pool. On parsing the message, the second thread determines that the message needs to be "patched" into an existing thread, wakes up the correlated thread and returns. Now let the thread pool size be **N**. If there are **N** concurrent requests, then all the **N** threads will be blocked waiting on the second `receive`. When a correlated message comes in, it will be put into the queue. However, since all the threads are blocked, it will never be removed from the queue and the system will go into a deadlock.

A simple solution is to spawn a new processing thread for each incoming request. In this scenario, the processing thread pool is not maintained and the system spawns a thread as and when required. This approach has major drawbacks. The time required to create and destroy the thread is more than the time required for allocating a thread from a thread pool. Further, in this situation, the container (servlet or EJB as the case may be) is not able to control the load on the system, and the number of threads in the system will grow arbitrarily as the load increases. This adversely affects performance as the system has limited CPU/memory resources and performance deteriorates when the number of threads become very large [23].

A second solution to the deadlock problem is to remove thread affinity. Whenever a composite service instance blocks on a corre-

lated `receive`, the engine saves its state and the processing thread is released. In this case, the processing threads are not blocked. This solution has the advantage that even though many *composite service instances* may be blocked on `receive`, no *thread* is blocked. This permits the system to serve another request and hence the system can make full use of the thread pool which is a limited resource. A serious drawback of this scheme is the increased processing time as the composite service instance state needs to be saved at each correlated `receive` and reloaded whenever it gets a correlated message. Another disadvantage is the increased memory requirement to save the states of all blocked composite service instances.

A third solution to the deadlock problem is to have separate processing thread pools for the message processing and message receiving threads. This way, the incoming request will be processed by a separate thread pool and the correlated message will be processed by a different thread pool. As soon as the correlated message receiving thread determines the composite service instance waiting for the message, it hands over the message to the blocked processing thread and the blocked thread resumes work. Since, the correlated message receiving thread is not doing much processing (it merely locates the composite service instance waiting for this message) only a small number of threads are required in this thread pool. Theoretically, only one thread in this pool is sufficient to break the deadlock. The disadvantage with this solution is that the processing threads are still blocked on the receive. As more and more threads are blocked, the throughput of the system goes down.

A fourth solution to the deadlock problem is to split the composite service partition such that the correlated `receive` and the remaining processing is moved to another composite service partition. This splitting can be done automatically during code partitioning. Hence, different composite service instances get created for incoming requests and for callbacks. Here, the initial composite service instance receives a request, does some processing, makes a request and exits. The correlated message is received by another composite service instance in another thread, which does the rest of the processing and sends the response back to the client if required. For example, **D0** of Figure 1(d) can be split into **D01** and **D02** as follows:

D01:

```
receive(client,c{name1,name2})
n1.name=c.name1
n2.name=c.name2
flow
    invoke(D1,n1{name})
    invoke(D2,n2{name})
end-flow
```

D02:

```
receive(D3,dir)
invoke(client,dir)
```

Thus, the BPEL4WS composite service partition is split into two separate partitions, such that neither has a callback `receive` within the partition. In this case, since the processing thread never blocks, there is no question of deadlock. Further, the throughput also improves as every thread is either busy doing useful work or waiting for a request and no thread is blocked, unable to do useful work. This scheme ensures optimal usage of system resources. However, this scheme has one disadvantage - context information related to the client needs to be propagated from the initial thread to the final split composite service instance along the call graph.

The first two solutions discussed here require changes to existing servlet container implementations. The first case requires the servlet container to create a thread per request instead of using a thread pool. The second solution requires the container or the BPWS4J engine to be able to swap a running process in a thread to be swapped out of the thread pool when it blocks on a callback and be swapped in again when it is ready to run. The remaining two solutions can be used without any modifications to existing systems. Hence, these solutions are preferable. We have evaluated the performance of the third and fourth solutions and the results are discussed in Section 4.

3.3 Error Propagation and Error Recovery

In decentralized orchestration, the entire state of the original composite web service is distributed across different nodes and this makes error recovery a complex task. It brings in issues related to transaction processing, which has been an active area of research and many efforts like WS-Transaction [7], BTP [4], Transactional Attitudes [20] are underway to solve this problem for centralized orchestration of composite web services. We restrict our discussion to problems that are peculiar to decentralized orchestration and propose a simple approach to solve error handling and recovery without concerning ourselves with transaction processing.

We discussed build time issues in fault handling and forward propagation of errors in Section 2 . However, in certain cases it might be easier and more reliable to propagate the error to a central entity that has some knowledge about the state of the entire composite service. The central entity can then stop all the composite service partitions that are currently executing. Knowing the state of the overall composite web service also helps in recovering from an error, as *undo* operations for all the activities that have already been completed need to be invoked. Thus, runtime monitoring of the individual composite service partitions on different nodes to calculate an estimate of the overall composite service progress is essential for error recovery and helpful for error propagation.

In our initial implementation, we have modeled this central entity as a *status monitor* which is implemented as a web service (refer Figure 2). On each node, a local monitoring agent runs that captures the local state of the composite service partition. A composite service consists of a list of activities and the status of the service is generally reported in terms of the status of those activities. This state can be retrieved either from the service database (either through polling at regular intervals or by setting triggers on the database) or through some API calls to the engine. The local monitoring agents periodically update the centralized status monitor that keeps track of the overall progress of the composite service instance using a process template that describes the composite service specification. The status monitor maintains the status of all the activities that were part of the original composite service, as well as the inputs and *undo* activities for all such activities. This forms a compensation list for the decentralized composite service. For one particular invocation of the composite service, the composite service partition instances that get created on different nodes share the same unique correlation id. The status monitor uses this correlation id to correlate the different partition instances on different nodes to one global composite service instance.

When an error occurs, the global state of the composite service maintained at the status monitor is used to determine all the activities that are still running and the activities that have already finished execution. If there are activities in the composite service that are still under progress and waiting for input from the failed activity, then the status monitor sends out an invalidation signal to all of them so that they stop executing and they undo the changes that they have already made. For all the activities that have already completed their execution, the status monitor should invoke the *undo* activities for all of them (using the compensation list) in their reverse order of execution.

We are in the process of conducting experiments to evaluate the overheads imposed by the error handling and status-monitoring framework and to determine the optimal frequency of status updates. Under high loads, maintenance of global state at one place can become a bottleneck and we are investigating other models of error recovery (forward error propagation, decentralized status monitoring) to alleviate this.

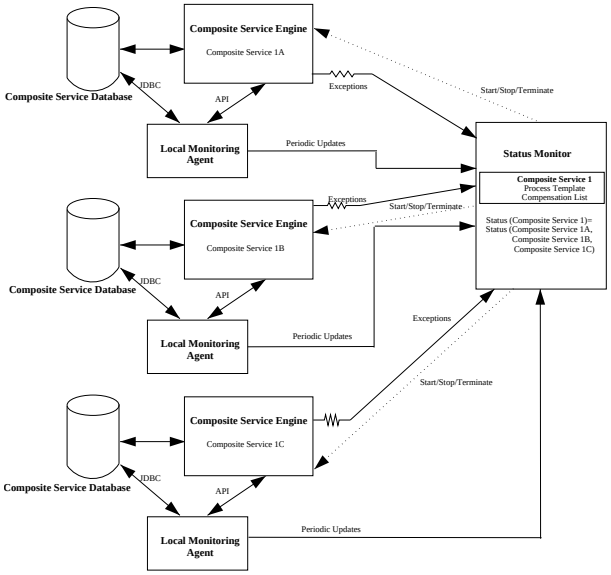


Figure 2: Status Monitoring

4. EXPERIMENTAL RESULTS

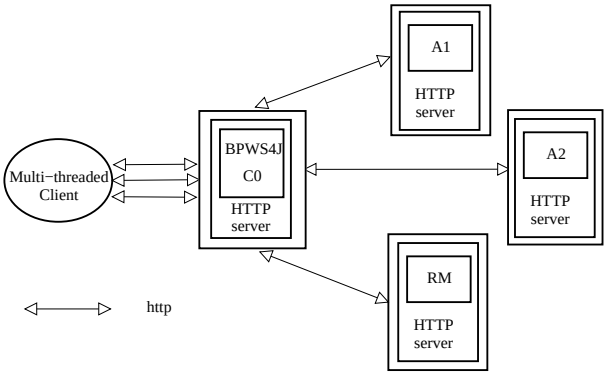


Figure 3: Experimental Set up for Centralized Orchestration

Experiments were conducted to study the performance of centralized and decentralized orchestrations using the *FindRoute* example discussed in section 2. Dummy implementations were used for services `A1`, `A2` and `RM` shown in Figure 1, and the processing (service time) was simulated by a *sleep*. We used the BPWS4J engine to orchestrate specifications written in BPEL4WS. A tool [16] was used to automatically partition the BPEL4WS code. The following different configurations were studied :

- The Centralized orchestration as shown in Figure 3.
- Decentralized orchestration with HTTP as the communication protocol between BPWS4J engines. We used "pseudo"-oneway messages, as discussed in section 3.1, for engine-to-engine communication. Two thread pools were used within a standard `HTTP` server to resolve the deadlock problem.
- Decentralized orchestration with JMS as the communication protocol between BPWS4J engines. In this case also two thread pools were used within the JMS container to resolve the deadlock problem.
- Decentralized orchestration with JMS as the communication protocol between BPWS4J engines and with partition `D0` split into two components as explained in Section 3.2.

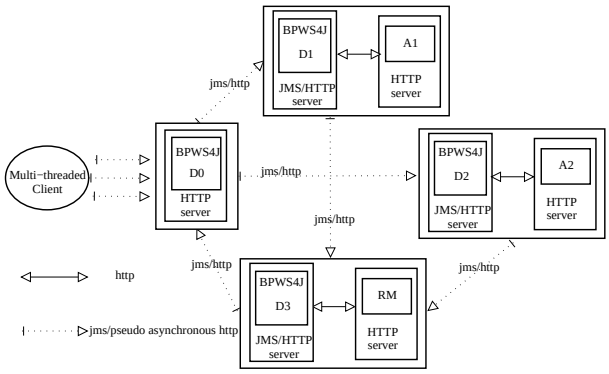


Figure 4: Experimental Set up for Decentralized Orchestration

The experimental set up consisted of four standard Intel-based machines running Linux and connected by a 100 Mb/s LAN. We used two multi-threaded asynchronous clients from two different machines to load the system. In centralized orchestration, the BPWS4J engine was hosted inside a standard `HTTP` server and each of the web services was deployed on a different machine on a `HTTP` server. In the decentralized-HTTP and decentralized-JMS orchestration, each of the BPWS4J engines (labeled `D0` to `D3` in Figure 4) was hosted on a different machine inside a HTTP or JMS server. The web services `A1`, `A2` and `RM` were co-located with `D1`, `D2` and `D3` respectively, but ran on different HTTP servers as shown in Figure 4. In the decentralized JMS set up with split, the two partitions - `D01` and `D02` (refer section 3.2), were hosted by the same server to share context information. In all the four configurations described above, the invocation of the actual web services was always SOAP over HTTP.

The system was loaded in three different ways - (i) by increasing the request rate, (ii) by increasing the size of the messages and (iii) by increasing the service time. We used the following parameters to load the system :

- Two multithreaded asynchronous clients using a total of 10 to 200 threads generated requests at a rate of 60 requests/minute to 1200 requests/minute.
- Service time at `A1`, `A2` and `RM` web services was varied from 500 ms to 8000 ms
- Size of the messages was varied from 512 bytes to 24 Kbytes.

We measured the average response time at the server (`C0` in figure 3 and at `D0` in figure 4) to serve the request and also the throughput (number of requests processed per minute).

Variation of Performance with Request Rate. The request rate variation experiments were conducted at a fixed message size of 512 bytes and service time of 4000 ms. Figure 6 shows the variation of throughput with request rate for the four orchestrations. At low request rates the throughput is the same for all the orchestrations since total number of concurrent requests executing in the system are still lower than the size of the thread pool. With increase in request rate, the centralized orchestration is the first to saturate and starts showing a drop in throughput as most of the threads are blocked waiting for a response from the component web services. The decentralized-HTTP and decentralized-JMS are the next to reach saturation. While the decentralized-JMS version degrades gracefully, the decentralized-HTTP version shows a rapid decline and eventually becomes worse than the centralized version. This is due to the inherent problems with pseudo asynchronous messaging over HTTP as discussed in Section 3.1. In decentralized-HTTP orchestration, we see a cumulative effect of using pseudo asynchronous messaging over HTTP at multiple places, particularly when the system is under high CPU load as the processing thread doesn't get scheduled for a long time. The throughput for decentralized-JMS declines as the load increases and eventually becomes similar to centralized orchestration as the thread pool at `D0` (refer Figure 4) eventually becomes the bottleneck and most of the threads are just blocked waiting for a callback from `D3`. The JMS-split orchestration scales the best as there are no threads blocked and all threads are doing useful work as discussed in Section 3.2.

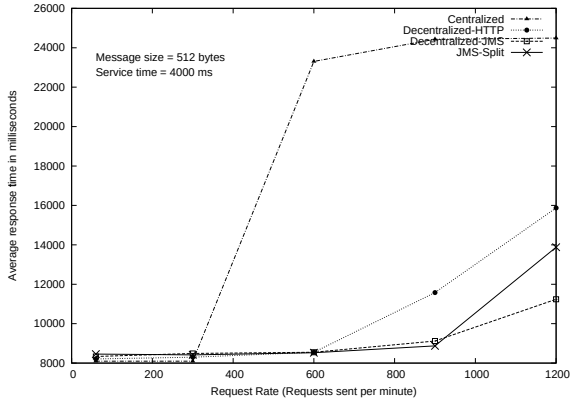


Figure 5: Performance Comparison: Response Time Variation with Request Rate

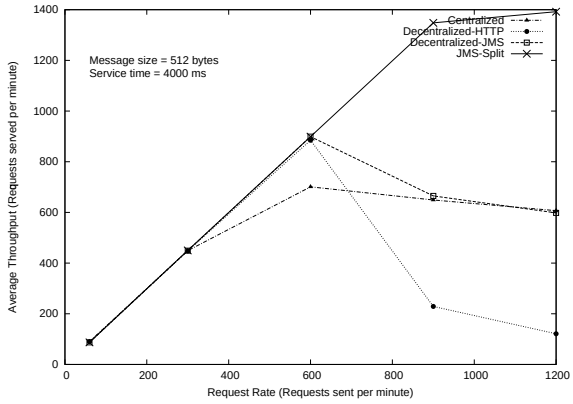


Figure 6: Performance Comparison: Throughput Variation with Request Rate

Figure 5 shows the variation of response time with request rate for the four orchestrations. Although, at low load, throughput of the decentralized-JMS and centralized orchestrations is the same, the response time of the centralized orchestration is slightly better than decentralized-JMS orchestration. Thus, overheads of decentralization make it perform worse than centralized orchestration at low loads. The response time of the JMS-split version is comparable with that of the decentralized-JMS version at lower loads, but deteriorates relatively at higher rates. With splitting the throughput increases as there are no blocking callbacks and no threads are blocked. At very high request rates, the decentralized orchestration using JMS-split allows a very large number of concurrent requests executing in the system, each of which are doing useful work and not just blocked waiting for a response (as in the case of decentralized JMS orchestration). This results in higher CPU load on the system and consequently, higher response times. Thus, at very high request rates there is a trade-off between throughput and response time in the two orchestrations.

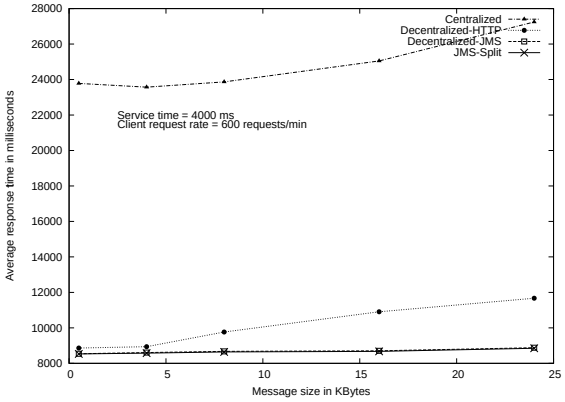


Figure 7: Performance Comparison: Response Time Variation with Message Size

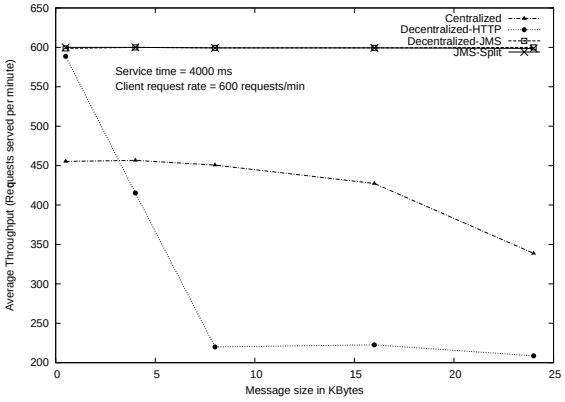


Figure 8: Performance Comparison: Throughput Variation with Message Size

Variation of Performance with Message Size. The message size variation experiments were conducted at a fixed request rate of 600 requests/min and service time of 4000 ms. Figure 7 and Figure 8 show the effect of message size on response time and throughput. The graphs clearly show that both response time and throughput degrade in the centralized orchestration when the

message size increases. There is also a clear difference in performance between the decentralized-HTTP and the decentralized-JMS orchestrations due to reasons mentioned in Section 3.1. There is no observable difference in the performance of the decentralized-JMS and the JMS-split orchestrations. This indicates that the total number of concurrent requests executing in the system is lower than the size of the thread pool and hence number of blocked threads have no impact on system performance in terms of response time and throughput.

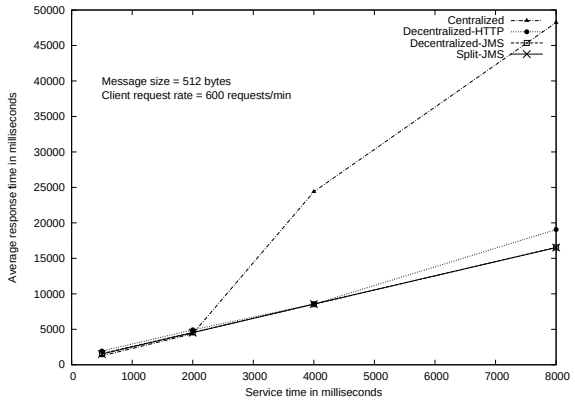


Figure 9: Performance Comparison: Response Time Variation with Service time

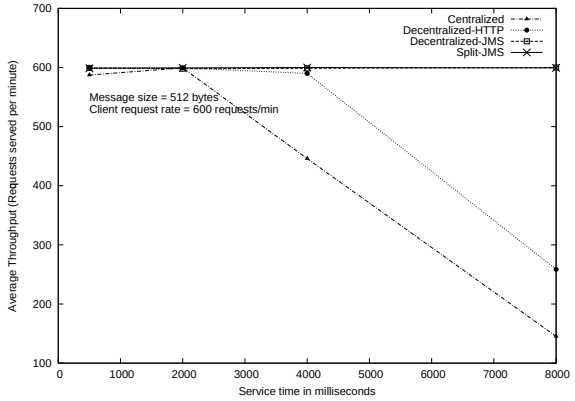


Figure 10: Performance Comparison: Throughput Variation with Service time

Variation of Performance with Service Time. The experiments on variation of service time were conducted at a fixed rate of 600 requests/min and a message size of 512 bytes. Figure 9 and Figure 10 show the effect of service time on response time and throughput. Here again, the experiments show that centralized orchestration does not scale with increasing service time. The decentralized-HTTP orchestration has response time that is comparable with that of decentralized-JMS but the throughput is comparatively very poor. At these loads also there is no difference in the performance of the decentralized-JMS and the JMS-split orchestrations. This again reflects the fact that the total number of concurrent requests executing in the system is lower than the size of the thread pool in these experiments.

Other Experiments. We conducted our experiments on a number of sample composite web services in addition to the *FindRoute* service mentioned in this paper. The results have not been mentioned in this paper for the sake of brevity. We observed that decentralization provides performance benefits even in cases where there was no inherent concurrency. The performance gain was mainly due to reduction of network traffic and distribution of computation across different nodes.

We also carried out preliminary experiments comparing decentralized orchestration with centralized orchestration using horizontally scaled load balanced servers. The total number of resources in both the cases were kept the same. We observed that although horizontal scaling and load balancing helps centralized orchestration perform better than decentralized orchestration at low loads, decentralized orchestration scales better at higher loads. This can be again attributed to optimal utilization of threads in a decentralized orchestration system as they are not blocked waiting for responses and allow large number of concurrent requests to be executed.

5. RELATED WORK

As web services become ubiquitous, a lot of effort is being spent in studying different ways of composing them to create more useful and complex services [6]. Four models of service composition have been proposed [14, 24] based on centralized and distributed flow of data and control messages between the services. The only two models that are relevant for composition of third party web services are the centralized control flow centralized data flow model (centralized orchestration) and distributed control flow distributed data flow model (decentralized orchestration).

The idea of using decentralized control of workflows and decentralized orchestration of web services has been proposed in earlier research. Benatallah et al. [6] describe a peer to peer execution pattern for orchestrating web services to overcome the bottleneck associated with having a centralized controller. The responsibility of coordinating the execution of a composite service is distributed across the providers which host the components of the composite service. However, [6] does not describe in detail the performance benefits, potential problems and various build time and runtime issues involved in this process.

Partitioning of the workflow specification using state and activity charts to enable distributed execution according to original semantics has been studied in[15]. A synchronization scheme that guarantees correct synchronization between workflow engines executing the partitions of a workflow is also developed. They also consider the issue of fault tolerance by providing exactly once semantics for delivering synchronization messages. Their approach is very similar to ours and a lot of interesting lessons can be learnt from their effort. However, they do not consider workflow-based composition of web services (which is our main focus) and the problems that are peculiar to this kind of composition.

Arjuna [19], a WFMS for CORBA-based environments, decentralizes both the workflow coordination and activity execution. This execution model decentralizes the coordination of a process by installing "taskcontroller" objects in different domains that execute and manage tasks and coordinate with each other to deliver workflow routing functionality.

The RainMan execution model [18] separates the responsibility of workflow coordination from activity execution by creating two classes of entities, Sources and Performers. In effect, while the coordination of each process remains localized within a Source object, the actual execution of activities is decentralized across a network of Performers over which Sources have very limited control. This model is essentially the centralized orchestration of web ser-

vices with web services acting as the Performers and the workflow acting as the Source.

A distributed workflow control architecture has been proposed in [12], where the agents schedule and coordinate the workflow instances. To execute a workflow instance, the agents that are responsible for executing the steps of that workflow instance have to communicate with each other transferring the entire state of the workflow from one agent to the other. The paper also describes how events should be propagated between the agents as efficiently as possible while still satisfying the failure handling and coordinated execution requirements.

A lot of work has also been done in understanding and solving various runtime issues in parallel and distributed systems. Decentralized orchestration relies heavily on asynchronous messaging. The benefits of asynchronous messaging, better throughput and graceful shutdown, have been studied in [11]. The importance of the design components, queues and thread pools for building highly concurrent systems, have been discussed in [23] and the benefits and trade-off in concurrency have been explored in [22].

In our earlier work [17] we have studied the issues of concurrency and synchronization in decentralization. There we describe an algorithm to identify different forms of concurrency in a composite service specification and considers the impact of dynamic binding and faults on synchronization constructs. In [16] we give an algorithm for partitioning the BPEL4WS programs.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have identified various build time and runtime issues in decentralized orchestration of composite web services and also discussed solutions to address them. We covered issues ranging from code partitioning for decentralization to detailed discussion of the servers that participate in decentralized execution - their thread pool design and communication protocols. We showed that the thread pool design can be a source of potential deadlocks and that JMS is a more efficient communication protocol than HTTP for engine-to-engine communication in a decentralized setup. We also discussed build and runtime issues in error handling and error recovery We presented an experimental evaluation of the performance of decentralization orchestration as compared to centralized orchestration in terms of throughput, average response time and scalability. Our results reconfirmed the performance benefits that decentralization provides. We also experimentally evaluated two different decentralization schemes and showed that at very high loads there is a trade-off between throughput and response time with the two schemes.

With proper design, optimal partitioning and run time support for error handling and recovery, decentralized orchestration provides an attractive approach for execution of complex high performance composite services.

Our current tool for code partitioning automatically generates partitions with the splitting optimization. We are working on enhancing the tool to automatically partition fault handlers and insert new fault handlers. We are also investigating various architectures for error handling and error recovery in a decentralized setup and quantify their effect on performance. In addition, we are working on infrastructure for dynamic reconfiguration (based on runtime monitoring) of composite services to obtain optimal performance. We plan to study the effect of deployment topologies on performance and work on various synchronization protocols for decentralization in future.

7. ACKNOWLEDGMENTS

The authors would like to thank Neeran Karnik and Sugata Ghosal from the IBM India Research Laboratory for their comments and suggestions during this work.

APPENDIX

A. SUMMARY OF BPEL4WS CONSTRUCTS AND NOTATION

Figure 11 gives the subset of BPEL4WS constructs used in this paper along with a description of each construct and the corresponding notations we have used.

B. REFERENCES

[1] Business Process Execution Language for Web Services Java Run Time (BPWS4J). http://www.alphaworks.ibm.com/tech/bpws4j.

[2] Business Process Execution Language for Web Services Version 1.1. http://www.ibm.com/developerworks/library/ws-bpel/.

[3] Enterprise Java Beans Specification (EJB) 2.1. http://java.sun.com/products/ejb/.

[4] OASIS Business Transaction Protocol, Committee Specification 1.0. http://www.oasis-open.org/business-transaction.

[5] A. Arkin, S. Askary, S. Fordin, W. Jekeli, K. Kawaguchi, D. Orchard, S. Pogliani, K. Riemer, S. Struble, P.Takaci-Nagy, I. Trickovic, and S. Zimek. Web Service Choreography Interface (WSCI) 1.0. http://wwws.sun.com/software/xml/developers/wsci/, 2002.

[6] B. Benatallah, M. Dumas, M. C. Fauvet, F. Rabhi, and Q. Z. Sheng. Overview of Some Patterns for Architecting and Managing Composite Web Services. In *ACM SIGecam Exchanges*, volume 3.3, pages 9–16, 2002.

[7] F. Cabrera, G. Copeland, B. Cox, T. Freund, J. Klein, T. Storey, and S. Thatte. Web Services Transaction (WS-Transaction). http://www-106.ibm.com/developerworks/webservices/library/ws-transpec/.

[8] D. W. Cheung, E. Lo, C. Y. Ng, and T. Lee. Web Services Oriented Data Processing and Integration. In *Proceedings of the Twelfth International World Wide Web Conference*, Budapest, Hungery, May 2003.

[9] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web Services Description Language (WSDL) 1.1. http://www.w3.org/TR/wsdl, March 2001.

[10] S. Graham, S. Simeonov, T. Boubez, G. Daniels, D. Davis, Y. Nakamura, and R. Neyama. *Building Web Services with Java: Making sense of XML, SOAP, WSDL and UDDI*. Sams; ISBN:0672321815, 2001.

[11] S. D. Gribble, E. A. Brewer, J. M. Hellerstein, and D. Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the Symposium on Operating Systems Design and Implementation(OSDI2000)*, October 2000.

[12] M. U. Kamath and K. Ramamritham. Pragmatic Issues in Coordinated Execution and Failure Handling of Workflow Control Architectures. Computer Science Technical Report 98-28, University of Massachusetts, August 1998.

[13] R. Khalaf, N. Mukhi, and S. Weerawarana. Service-Oriented Composition in BPEL4WS. In *Proceedings of the Twelfth*

BPEL4WS construct	description	notation
Control Flow Constructs		
`sequence`	sequential flow	`sequence` ... `end-sequence`
`switch`	conditional flow	`switch` ... `end-switch`
`while`	iterative flow	`while` ... `end-while`
`pick`	non-deterministic conditional flow	`pick` ... `end-pick`
`flow`	concurrent flow similar to cobegin-coend	`flow` ... `end-flow`
`link`	wait-notify type of synchronization	`source(linkId)`, `target(linkId)`
`throw`	throws an exception	`throw(faultName)`
Data Structures		
`variable`	variables include a set of parts analogous to fields	`variableName{part1, part2, ... partn}`
Activities		
`invoke`	synchronous (blocking) invocation on a parter **P**, sending data from an input variable **in** and receiving the response in the output variable **out**	`invoke(`**P**, **in**, **out** `)`
`invoke`	asynchronous (oneway, nonblocking) invocation on a parter **P**, sending data using an input variable **in** (no response variable)	`invoke(`**P**, **in** `)`
`receive`	blocking receive of data from a partner **P** into a variable **var**	`receive(`**P**, **var**`)`
`reply`	send response to a partner **P** from a variable **var**	`reply(`**P**, **var**`)`
`assign`	assignment	`var1.p1 = var2.p2`
`catch`	handles an exception	`catch(faultName)`

Figure 11: BPEL constructs and notations

International World Wide Web Conference, Budapest, Hungery, May 2003.

[14] D. Liu, K.H.Law, and G. Wiederhold. Analysis of Integration Models for Service Composition. In *Proceedings of the third international workshop on Software performance*, Rome, Italy, July 2002.

[15] P. Muth, D. Wodtke, J. Weissenfels, D. A. Kotz, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems (JIIS)*, 10(2), 1998.

[16] M. G. Nanda, S. Chandra, and V. Sarkar. Decentralizing Composite Web Services. In *Proceedings of Workshop on Compilers for Parallel Computing*, January 2003.

[17] M. G. Nanda and N. Karnik. Synchronization Analysis for Decentralizing Composite Web Services. In *Proceedings of the ACM Symposium on Applied Computing(SAC)*, Melbourne, FL, October 2003.

[18] S. Paul, E. Park, and J. Chaar. RainMan: A Workflow System For The Internet. In *Proc. Usenix Symposium on Internet Technologies and Systems, California*, December 1997.

[19] F. Ranno, S. K. Shrivastava, and S. Wheater. A System for Specifying and Coordinating the Execution of Reliable Distributed Applications. In *International Working Conference on Distributed Applications and Interoperable Systems (DIAS'97)*, September 1997.

[20] I. R. T. Mikalsen, S. Tai. Transactional Attitudes: Reliable Composition of Autonomous Web Services. In *Proceedings of Workshop on Dependable Middleware-based Systems*, June 2002.

[21] W. M. van der Aalst. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In *Business Process Management*, pages 161–183, 2000.

[22] M. Welsh, D. Culler, and E. Brewer. SEDA: An Architecture for Scalable, Well-Conditioned Internet Services. In *Proceedings of 18th ACM Symposium on Operating Systems Principles(SOSP'01)*, Canada, October 2001.

[23] M. Welsh, S. Gribble, E. Brewer, and D. Culler. A Design Framework for Highly Concurrent Systems. CS Technical Report UCB/CSD-00-1108, University of California, Berkeley, October 2000.

[24] A. Yew, A. Strand, A. Liotta, and G. Pavlou. Aggregation of Composite Location-Aware Services for Mobile Cellular Networks. In *Proceedings of 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, Germany, October 2003.