An Event-Driven Pattern for Asynchronous Invocation in Distributed Systems

Soheil Toodeh Fallah, Ehsan Zaeri Moghaddam, and Saeed Parsa

Iran University of Science and Technology, Tehran, Iran

Abstract
Asynchronous invocation reduces the average running time of distributed programs by providing concurrency mechanisms. The fact of occasionally having to check for return values in calling asynchronous methods is a noticeable drawback in such systems. We can cope with this issue by making instructions dependent on the return values of asynchronous methods as appropriate listener threads. In this paper, we have proposed a pattern for asynchronous invocation in order to enhance the client's performance in distributed systems. The layered model of the proposed pattern led us to a middleware-independent framework. The evaluation results indicate that our solution shall introduce a unified pattern for asynchronous remote method invocation.

Keywords
Asynchronous Invocation, Pattern, Distributed Systems, Remote Method, Service, Framework

1. Introduction

Asynchronous remote calls enhance performance of distributed code, by allowing concurrent execution of the distributed components. Dependency of the statements which are being executed after a remote asynchronous invocation to the values affected by the callee may be a barrier against concurrent execution of the caller and the callee. To resolve the difficulty, statement reordering algorithms may be applied to increase the distance between the call statement and the very first positions where the results of the call are required [4,14,15,16]. A major difficulty in applying reordering algorithms is necessity to predict the execution time of the program code statically [13]. These algorithms are static and do not consider the runtime behavior of the code to be reordered.

In order to achieve maximum concurrency in the execution of distributed code, execution of the statements which are dependent on any value affected by an asynchronous call statement is delayed by inserting them, at run time, into a separate thread which could be executed when the results of the remote call are required. This approach is offered in a framework introduced by Zdun [3]. An important problem is that in Zdun framework asynchronous method invocation are restricted to web service technology. Also, in Zdun framework the caller has to wait for the results of an asynchronous call by applying a *busy waiting* method. As described in Section 3.1 in the approach proposed in this paper, notification of completion of asynchronous calls is sent to the callers through events raised by the callee. In addition, presenting a layered architecture for asynchronous remote invocations has made it possible to apply any middleware supporting remote calls, within our proposed framework.

Asynchronous invocations are not directly supported by the conventional programming languages. There are several middlewares such as Apache Axis2 [7,8,9], CORBA and Java symphony [6] which provide their own interfaces and libraries to support remote asynchronous calls. However, a major difficulty is that there are no standard interfaces for asynchronous method invocations [2,3,5,11,12]. To resolve the difficulty, in Section 2, a layered architecture is proposed for remote asynchronous calls.

The design of our framework for asynchronous invocations is centered on a design pattern depicted in Figure 2. This pattern is represented as a class diagram including all the classes required to apply asynchronous invocations independent of any underlying middleware interfaces and communication protocols. There is an interface class, within this pattern, which allows the callee to raise events for notifying the caller when the results of the invocation are ready.

The remaining parts of this paper are organized as follows. A design pattern for asynchronous calls is introduced in Section 2. Section 3 describes the design and implementation of a framework for asynchronous invocations. A practical evaluation of the speed and the size of the code required to implement a worked example within the proposed framework is presented in Section 4. The conclusions and recommendation for future improvement of the proposed approach is presented in Section 5.

2. ED-AMI Pattern

As mentioned above, middleware or platform dependency, large set of instructions required for asynchronous invocations and the lack of a well-accepted standard, have made it difficult to develop distributed programs.

Listing-1 represents an example of a typical asynchronous call without the details of accessing middleware. In this listing both the methods, f and g are invoked asynchronously.

```
    ….
            a=f(x,y);    // 1st remote asynch. call point
     ….
A:   while (a is not ready){ } //wait for the 1st call
     b=a*2;      // Use point
     c=g(t,p);   // 2nd  remote asynch. call point
     ….
B:   while (c is not ready) { } // wait for the 2nd call
     d=c+5;      //Use point
      ….
```

Listing-1

In Listing-1, there are two *busy waiting while loops* labeled A and B. These loops terminate when the return values of the asynchronous calls are ready. However, there may be instructions, which are independent of the call results and may execute in parallel with the called methods, f and g.

In Listing 2 all those instructions in Listing 1 which are, directly or indirectly, data dependent on the results of the asynchronous calls to methods f and g are threaded. These threads are executed whenever their required data is ready. After the data dependent instructions are threaded there will be no need to wait for the results of asynchronous calls and the waiting loops may be removed from within the program code.

```
….
AsyncResult a = threadExecutionOf ( f(x,y) );
….
AsyncResult b= threadExecutionOf (a*2);
AsyncResult c= threadExecutionOf ( g(t,p) );
….
AsyncResult  d= threadExecutionOf ( c+5 );
….
```

Listing-2

Apparently, multi-threading itself incur excessive runtime overheads. To alleviate the overheads, a thread pooling [10] technique can be applied.

As mentioned, simplifying asynchronous mechanism and creating a middleware independent approach was another consideration during designing our pattern.

To achieve this goal, we recommend a pervasive pattern, which is applied in a layered architecture that is illustrated in Figure 1.

In this architecture, service or remote method caller layer is separated from middleware-dependent invocation implementation layer by using AMI layer that is a framework based on ED-AMI pattern.

Developer implements method invocation in service handler layer and then uses that implementation in method caller layer using ED-AMI pattern.

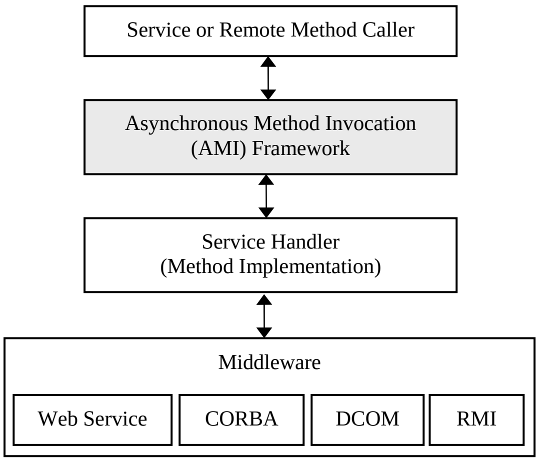

Service or Remote Method Caller
Asynchronous Method Invocation (AMI) Framework
Service Handler (Method Implementation)
Middleware
Web Service
CORBA
DCOM
RMI

Fig. 1 Proposed Layered Architecture

Listing-3 and Listing-4 represent asynchronous invocation in Apache Axis 2 and Java Symphony. The samples show that each framework provides different way of asynchronous invocation, which is specific to its approach.

```
OMElement payload=…
Call call=new Call();
call.setTo(
            new EndpointReference(
                 AddressingConstant.WSA_TO,"http://...")
            );
call.setTransportInfo(
            Constants.TRANSPORT_HTTP,
            Constants.TRANSPORT_HTTP,false);
Callback callback=new Callback() {
        public void onComplete(AsyncResult result){
            //What user can do to result
        }
        public void reportError(Exception e) {
            //on error
        }
};
call.invokeNonBlocking(
        operationName.getLocalPart(),
        payload,callback);
```

Listing-3

```
JSObject obj = new JSObject("ClassName");
//invoke remote method with parameters;
Object[] params = {new Param1(), new Param2()};
Class[]  paramTypes = new Class[] {
                    Param1.getClass(),
                    Param2.getClass() };
                    ResultHandle handle =
  obj.ainvoke("methodName",params [,paramType]);
.....
//**** verify whether result is available
if (handle.isReady()) {
   // wait for result to arrive in blocking mode
   ResultClass result =
                    (ResultClass)handle.getResult ();
}
```

Listing-4

The structure of our proposed pattern is depicted in Figure 2. The collaboration of these classes and interfaces will be described in Section 3.

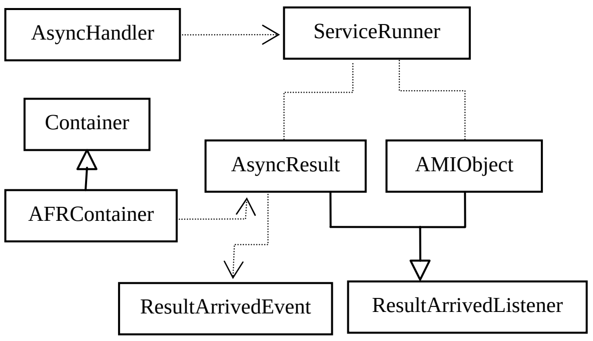


Fig. 2 Structure of Proposed Pattern

Following code illustrates a sample of asynchronous invocation using AMI.

```
AsyncResult resultA=
            AsyncHandler.invoke (
            "iust.servicehandler.ServiceA","getResult",cityId);
```

Listing-5

By executing this statement, *getResult* method from *ServiceA* class in *iust.serviechandler* package execute in a separated thread. *getResult* can be implementation of a web service invocation or CORBA remote method call, which need *cityId* as its input parameter. As you can see, there is no middleware dependency in the caller side.

Result value will be returned to caller using an event driven approach that is illustrated as follows:

```
public void resultArrived(ResultArrivedEvent e){
      try{
            ((AsyncResult)e.getSource()).getResult();
          }
      catch (ServiceInvocationException ex){
            //handle exception
          }
}
```

Listing-6

Finally, according to features like middleware-independency and event driven architecture and also simplifying asynchronous invocation, the proposed approach can be considered as an standard pattern in different environment.

In other words, ED-AMI pattern can be applied in the following conditions:

- When software is developed in a distributed environment using a Grid or Service-oriented architecture.
- While two or more remote methods can be invoked interchangeably and the fastest result is used. (We will discuss this matter in Section 3.1)
- When the results of different services can be sent separately for user in an interactive environment.
- When developers seeking an approach, which is efficient and still simple and easy to use.

In the next section, we will describe our proposed framework and its structure.

3. The Proposed Framework

In this section, we will introduce a framework based on ED-AMI pattern. Figure 3 depicted the structure of framework components in the form of layered architecture that is shown in Figure 1.

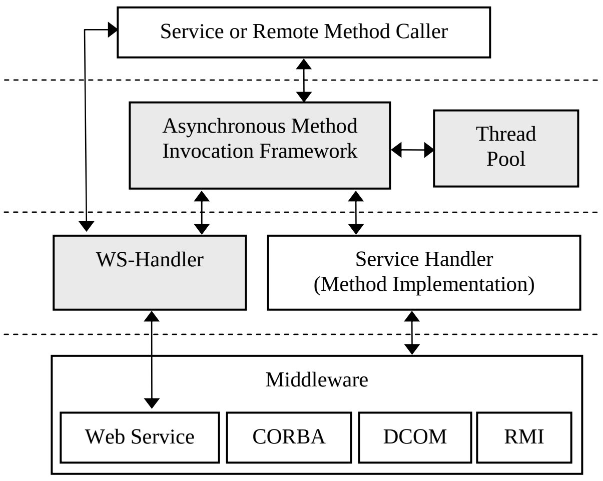

Service or Remote Method Caller
Asynchronous Method Invocation Framework
Thread Pool
WS-Handler
Service Handler (Method Implementation)
Middleware
Web Service
CORBA
DCOM
RMI

Fig. 3 The Structure of Proposed Framework

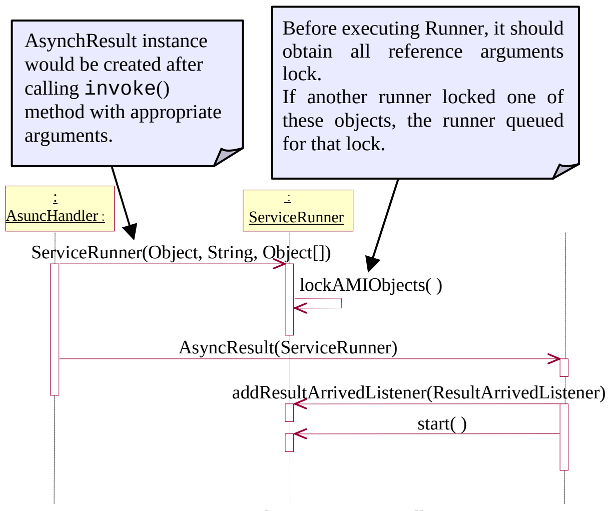
Thread Pool and WS-Handler are two components added to the main part to enhance and extend our approach.

The reason of using thread pool is discussed in Section 2. But WS-Handler is another supplementary and extendable component which provides some facility to invoke web services.

3.1. Collaboration

The sequence of operations execution is described in this section. An important note is that our approach provides some more features such as policy definition and passing call-by-reference arguments to service invocations.

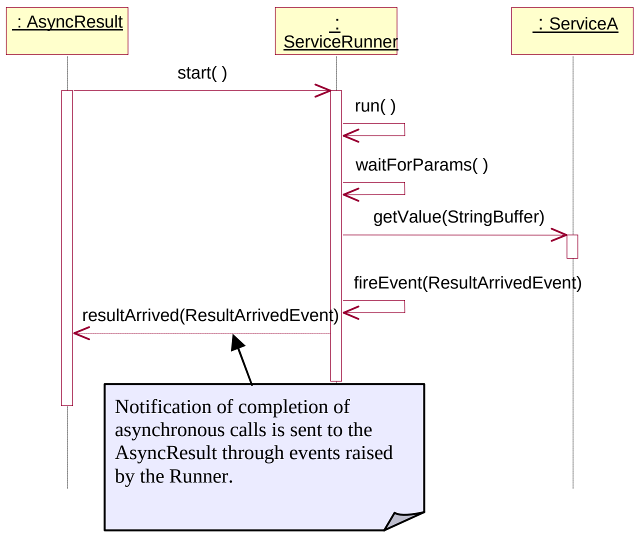
As shown in Figure 4, the requester calls static method *invoke()* from *AsyncHandler* class in order to create an instance of *ServiceRunner* and obtain locks for reference arguments. If one of these objects is already locked by another thread, this runner is added to the waiting queue of that object. *AsyncHandler* class is the beginning point of asynchronous invocation and *ServiceRunner* is a thread which is responsible for parallel execution of service or method invocation. An *AsyncResult* instance is created and the runner assigns to it. In order to complete this relation between *AsyncResult* instance and the runner, the instance registers itself as a listener in *ServiceRunner*, consequently, *AsyncResult* is informed as soon as results are available.



AsynchResult instance would be created after calling invoke() method with appropriate arguments.
Before executing Runner, it should obtain all reference arguments lock.
If another runner locked one of these objects, the runner queued for that lock.
: AsuncHandler :
: ServiceRunner
ServiceRunner(Object, String, Object[])
lockAMIObjects( )
AsyncResult(ServiceRunner)
addResultArrivedListener(ResultArrivedListener)
start( )

Fig. 4 Initializing AsyncHandler

The operations in the caller side continue while *invoke()* result, which is an instance of *AsyncResult* class, returns to main thread (caller) and then, service runner executes method in parallel. As illustrated in Figure 5, to start the execution of invoked method, it is needed to check arguments type. If the argument is an *AsyncResult* instance and its result is not ready yet, the current service runner should wait until all parameters values become available, and then starts the execution. At the end of execution, the notification of completion is sent to the caller by raising an event.



: AsyncResult
: ServiceRunner
: ServiceA
start( )
run( )
waitForParams( )
getValue(StringBuffer)
fireEvent(ResultArrivedEvent)
resultArrived(ResultArrivedEvent)
Notification of completion of asynchronous calls is sent to the AsyncResult through events raised by the Runner.

Fig. 5 Executed Instructions From Beginning Until The End of Invocation

Up to here, we have discussed the main sequences of Asynchronous method invocation. However, as mentioned before, arguments passed to an asynchronous invocation can be an instance of AsyncResult, *AMIObject* or *Container*. The proposed approach employing the above types to make it possible to control every implicit and explicit relation between method invocations.

AMIObject is a class that facilitates the process of sending a call-by-reference argument to more than one invocation. There is a lock and a waiting queue in this class to manage all requests for obtaining lock from invocation runners. This class handles requests itself and creates a queue from requesters (service runners) and chooses next requester and notify it to begin execution while receives the notification of completion from last runner which owns the lock.

The *Container* interface makes it possible to define special policies for a set of invocations. A typical *Container* contains some asynchronous invocations which are invoked by a requester that considered some predefined conditions for their return values. For example, the *AFRContainer* class which is an implementation of *Container* interface, returns the most ready response from among a collection of services or methods invocations contained.

Figure 6 shows the sequence diagram of the method *waitForParams* from *ServiceRunner* class. The following operations will be done according to parameters passed to this method.

If the parameter is an instance of *AsyncResult*, the current runner links itself to the runner of this instance to continue execution after the execution of *AsyncResult* runner ended (State A in diagram).

If the passed parameter is a type of *Container* interface, the current thread waits to receive the *Container* result, which is based on a defined policy (State B).

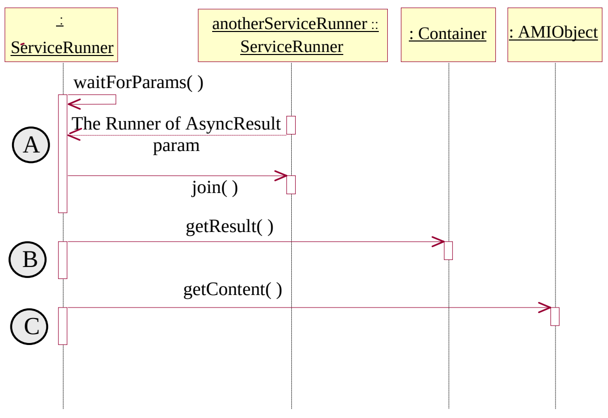


Fig. 6 Waiting For Passed Parameters to Invocation

Finally, if the parameter were an instance of *AMIObject* the current runner can get its content after obtaining its lock (State C).

When all parameters value become available to runner thread, the *waitForParams()* method ended and the runner can continue execution.

3.2. Implementation

The proposed structure for developing AMI-Framework has some essential points, which must be considered.

First of all, developing environment must support multi threading programming due to our proposed approach which is based on *ServiceRunner*.

As the reason for using events in framework implementation, supporting event driven programming is another necessary feature for developing environment. Also it is possible to use callback interface for environment that doesn't support events (Observer pattern [1, 12]).

As mentioned in Section 2, we use thread pooling technique to improve performance and minimize the overhead of using multi-threading. This is very important, because if we do not use this technique, it is possible that the developer invokes too many services and causes a very poor performance because of execution of a large number of threads.

Another problem in implementing *AMIObject* class is to control threads status, owning a lock, to be alive. Note that if the runner obtains a lock and then interrupted for any reason, other runners, which are waiting for that lock, will be in wait forever. In addition, while implementing *lockAMIObject()* method in *ServiceRunner* class, it is very important to check all needed locks and execute invocation only if all locks are available. This is a conservative approach to prevent deadlock.

4. Evaluation

In this section, we use execution time optimization, reduction of the number of statements needed for an asynchronous invocation and middleware independency of invocation instructions as our evaluation parameters in the form of an example.

We assumed that seven services named A to G, are invoked by a requester, asynchronously. Figure 7 illustrated these services and their dependency.

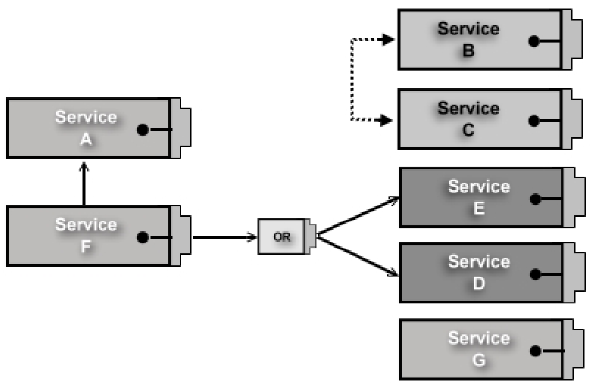


Fig. 7 Invoked Services and Their Dependency

The dependency between B and C is from a shared reference argument passed in their invocations. Service F is dependent to Service A result and the earliest return value of Service D and E. Service A and G are independent in their execution. Table 1 shows the execution time of these services.

Table 1: Services Execution Time

T_A	T_B	T_C	T_D	T_E	T_F	T_G
5ms	14ms	3ms	7ms	11ms	10ms	8ms

The implementation of our example using ZDun framework is illustrated in Listing-7.

```
AsyncRequester rA = new AsyncRequester();
PollObject pA = (PollObject) new SimplePollObject();
rA.invoke(pA, null, endpointURL,operationName, null, rt);

AsyncRequester rB = new AsyncRequester();
PollObject pB = (PollObject) new SimplePollObject();
rB.invoke(pB, null, endpointURL,operationName, null, rt);

AsyncRequester rD = new AsyncRequester();
PollObject pD = (PollObject) new SimplePollObject();
rD.invoke(pD, null, endpointURL,operationName, null, rt);

AsyncRequester rE = new AsyncRequester();
PollObject pE = (PollObject) new SimplePollObject();
rE.invoke(pE, null, endpointURL,operationName, null, rt);

AsyncRequester rG = new AsyncRequester();
PollObject pG = (PollObject) new SimplePollObject();
rG.invoke(pG, null, endpointURL,operationName, null, rt);

while (!pB.resultArrived()) {}   //Busy wait
Object res=pB.getResult();
AsyncRequester rC = new AsyncRequester();
PollObject pC = (PollObject) new SimplePollObject();
rC.invoke(pC, null, endpointURL,operationName, res, rt);

while (!(pA.resultArrived() &&
        (pD.resultArrived() || pE.resultArrived() ))) {}
AsyncRequester rF = new AsyncRequester();
PollObject pF = (PollObject) new SimplePollObject();
Object[] obj;
if (pD.resultArrived)
            obj={pA.getResult(),pD.getResult()};
else
            obj={pA.getResult(),pE.getResult()};
rF.invoke(pF, null, endpointURL,operationName, obj, rt);

// Use final results
```

Listing-7

The example implemented by our proposed framework is presented in Listing-8.

```
AsyncResult resA=AsyncHandler.invoke(
    new WSHandler(endPointURL,operationName,rt));
AMIObject ami=new AMIObject(new Object());
AsyncResult resB=AsyncHandler.invoke(
    new WSHandler(endPointURL,operationName,rt),ami);
AsyncResult resC=AsyncHandler.invoke(
    new WSHandler(endPointURL,operationName,rt),ami);
AsyncResult resD=AsyncHandler.invoke(
    new WSHandler(endPointURL,operationName,rt));
```

```
AsyncResult resE=AsyncHandler.invoke(
    new WSHandler(endPointURL,operationName,rt));
AFRContainer container=new AFRContainer();
container.add(resD);
container.add(resE);
Object[] obj={resA,container};
AsyncResult resF=AsyncHandler.invoke(
    new WSHandler(endPointURL,operationName,rt),obj);
```

Listing-8

Table 2 shows the invocation results. As shown in Table 2, AMI-Framework offers a better execution time and the minimum line of code, while it is applicable in different platforms and middlewares.

Table 2: Comparison of Results of Different Frameworks

	Sequential	ZDun	Axis2	AMI-Impl
Execution Time (ms)	47-51	24	17	17
Line of Code	7	29	32	12
Middleware independent	-	No*	No	Yes

* *It should be noted that ZDun solution is applicable just in Web Service environment and it does not support other technologies. In addition, this approach is protocol independent.*

5. Conclusion and Future Work

In this paper, a new approach for asynchronous method invocation is presented. This pattern is applicable in distributed systems because of the importance of asynchronous invocations which would lead to a better performance specially for the service requester. In addition to asynchronous invocation, the proposed pattern supports the definition of particular policies and parameter access control. ED-AMI pattern separates service invocation from its implementation which would facilitate extensibility in the form of flexible and distinct layers.

There are many open areas to cater for as our future work. Definitely, the employment of dependency graph in order to recognize dependant statements would optimize the current paradigm. This shall be accomplished through some sort of precompliler or runtime analyzer. We have also decided to develop this framework under .Net platform. Finally, the ED-AMI pattern can be extended so that it provides additional management layers to monitor system performance and enhanced traceability.

References

[1] Gamma. Eric, et al, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison Wesley Lonjman, Inc. 1998

[2] D. Schmidt, M. Rohnert, H. Buschmann, *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000

[3] ZDun. Uwe, et al, "*Pattern-Based Design of an Asynchronous Invocation Framework from Web Services*", International Journal of Web Service Research, Volume 1, No. 3, 2004

[4] S.Parsa, O.Bushehrian, "*Automatic Translation of Serial to Distributed Code Using CORBA Event Channels*", Lecture Note in Computer Science(LNCS), Vol.3733, pp. 152-161, Springer-Verlag, 2005

[5] M.Voelter, et al, "*Pattern for Asynchronous Invocation in Distributed Object Frameworks*", In proceedings of EuroPlop, Germany, 2003

[6] Th.Fahringer, *Java Symphony*, http://www.dps.uibk.ac.at/projects/javasymphony/, 2005

[7] S.Srinath, A.Ranabahu, *"Axis2-Future of Web Services*", Jax Magazine, Jun 2005

[8] S.Perera, A.Ranabahu, "*Web Services Messaging with Apache Axis2: Concepts and Techniques"*, ONJava.com, Jul 2005

[9] The Apache Software Foundation, *Apache Axis2*, http://ws.apache.org/axis2/ , 2005

[10] J.Heaton, *Creating a Thread Pool with Java*, SAMS, 2003

[11] H.Adams, *Asynchronous operations and Web services*, IBM, Jun 2002

[12] D.C.Schmidt, *Monitor Object: An Object Behavorial Pattern for Concurrent Programming*, Washington University, Department of Computer Science, 1999

[13] J.Hennessy, D.Patterson, *Computer Architecture: A Quantitative Approach (Third Edition)*, Morgan Kaufmann Publishers, 2003.

[14] A. Alet´a, et al, *Exploiting pseudo-schedules to guide data dependence graph partitioning*. In Proceedings of the 2002 International Conference on Parallel Architectures and Compilation Techniques, Charlottesville, VA, Sep 2002

[15] M.C.Golumbic, V.Rainish, *Instruction scheduling beyond basic blocks*. IBM 1. RES. DEVELOP. VOL. 34 NO. 1, Jan 1990

[16] M.Hagog, A.Zaks, *Swing Modulo Scheduling for GCC*, GCC Developers' Summit, 2004

Soheil Toodeh Fallah received the BS in Software Engineering from Azad University, Tehran Central Branch, Iran, and the MS degree in information technology from Iran University of Science and Technology. His research interests include middleware, distributed systems and software engineering. He is a member of Computer Society of Iran.

Ehsan Zaeri Moghaddam received the BS in Software Engineering from Azad University, Sari Branch, Iran, and the MS degree in information technology from Iran University of Science and Technology. His research interests include semantic web, distributed systems and HCI.

Saeed Parsa received the BS in mathematics and computer Science from Sharif University of Technology, Iran, and the MS and PhD degrees in computer science from the University of Salford at England. He is an associated professor of computer science at Iran University of Science and Technology. His research interests include software engineering, soft computing and algorithms.