The Architecture of an Event Correlation Service for Adaptive Middleware-based Applications

Yan Liu¹, Ian Gorton² and Vinh Kah Lee¹
¹National ICT Australia, Australia
²Pacific Northwest National Laboratory, U.S.A
jenny.liu@nicta.com.au; vinkkah.lee@nicta.com.au ; ian.gorton@pnl.gov

Abstract

Loosely coupled component communication driven by events is a key mechanism for building middleware-based applications that must achieve reliable qualities of service in an adaptive manner. In such a system, events that encapsulate state snapshots of a running system are generated by monitoring components. Hence, an event correlation service is necessary for correlating monitored events from multiple sources. The requirements for the event correlation raise two challenges: to seamlessly integrate event correlation services with other services and applications; and to provide reliable event management with minimal delay. This paper describes our experience in the design and implementation of an event correlation service. The design encompasses an event correlator and an event proxy that are integrated with an architecture for adaptive middleware components. The implementation utilizes the Common Based Event (CBE) specification and stateful Web service technologies to support the deployment of the event correlation service in a distributed architecture. We evaluate the performance of the overall solution in a test bed and present the results in terms of the trade-off between the flexibility and the performance overhead of the architecture.

Keywords: Event correlation, middleware, common based event, Web services.

1. Introduction

Distributed applications are increasing in complexity due to rapid changes in business requirements and the constant evolution of the technologies they are built from. Component-based middleware platforms have been used to address such challenges by providing a standardized service layer to support the development and deployment of distributed, server-side applications. Consequently, incorporating adaptive capabilities into middleware platforms is attracting considerable attention as a means to respond to both the functional and environmental changes that occur after application deployment [16].

Dynamic adaptation of middleware-based applications relies on constantly monitoring the system and adjusting configuration parameters to affect behavior. In order to realize the adaptation requirements, a set of control components analyse the data collected, and make decisions on optimal configuration settings, such as the server thread pool size, transaction time-out duration, the maximum number of concurrent connections[10], and so on.

In an adaptive application, monitoring activities typically generate data as events. These events encapsulate snapshots of runtime states of the application and middleware platform. In order to gain meaningful representation of these events for further analysis and decision making, events arriving from different sources must be managed and correlated to create a composite state of the running system. As a result, an efficient and flexible event correlation service is essential to ensure the scalability and performance of adaptive middleware-based applications.
One solution to achieve the goal of event correlation on a middleware platform is to implement the event correlation functionality as a platform service. The resulting architecture should allow the event correlation service to plug into the middleware platform without requiring changes to be made to the applications running on the middleware. Such a separation of concerns results in a major benefit: it reduces software development and maintenance costs because the complexity of special-purpose event correlation is factored out of the application components. In addition, it increases the flexibility in the event correlation service, as complex event properties and dependencies can be explicitly expressed in correlation rules that can be accessed and updated at runtime. Hence, this solution can effectively augment the capability of middleware platforms.

With the increasing adoption of service-oriented architectures (SOA), it is important that the event correlation functionality should be accessible as a service. This facilitates event correlation service interoperation and communication with various services and applications in a distributed computing environment.

Achieving the above goals presents two main challenges: (1) to seamlessly integrate the event correlation service with other services or applications in a distributed computing environment; and (2) to reduce the overhead of event correlation to avoid adverse degradation of overall application performance.

In our previous work, we have designed and implemented a configurable event correlation engine [11] and integrated it with adaptive J2EE-based applications on a single Java server. In this paper, we further expand our research to develop an event correlation service and integrate it in a highly distributed, loosely coupled environment. The event correlation engine built in our previous work [11] forms the base of the event correlation service. In order to address the above challenges, we significantly extend the event correlation engine to utilize the Common Based Event (CBE) standard and stateful Web services.

This paper first describes the core components of the event correlation engine and then presents the design and implementation utilizing Web services and CBE in order to operate in a distributed environment. We also empirically evaluate the overheads of the architecture by examining the performance characteristics of the major components in this architecture. This provides us with an understanding of the architectural trade-offs based on the performance and flexibility needs of the event correlation service.

The main contributions of this paper are:

- It describes an architecture for an event correlation service.
- It demonstrates that the event correlation functionality can be implemented using standards and generic middleware technologies.
- It identifies performance issues and summarizes the lessons learned from the design and implementation of service-oriented event correlation.

2. Related Work

Event correlation can be categorized by five aspects according to [2]. These include:

- **Compression-based correlation** monitors multiple occurrences of the same event, removes redundancies and reports them as a single event.
- **Threshold-based correlation** has a threshold to trigger a report when a specified number of similar events occur.
- **Filter-based correlation** inspects each event to determine if it matches a pattern defined by a regular expression. If a match is found, an action may be triggered as specified in the rule.
• **Sequence-based correlation** helps to establish causality of events. Events can be correlated based on specific sequential relationships. For example, synchronizing multiple events such as event A being followed by event B to trigger an action.

• **Time-based correlation** is useful for correlating events that have specific time-based relationships. Some problems can be determined only through such temporal correlation. For example, time based correlation can be used to implement cleanup rules given a specific interval.

Event correlation technologies have been widely used to monitor and analyse enterprise-wide event data for a variety of applications. There are many event correlation approaches to specific application domains including distributed applications, network communication and intrusion detection. Most of these techniques assume access to the raw event data and they execute rules specified in expression languages for correlating and filtering events.

Luckham et al. propose an object oriented language Rapide for describing event patterns to determine event causalities and root cause analysis [12]. Appleby et al describe techniques for high-speed event correlation to manage network applications [1], which decompose events into codes in bit vectors and enable high-speed lookup operations.

Research efforts have also focused on event algebras for active databases and reactive systems [22]. An event correlation model from Stanford University [17] provides a rigorous formal model of event correlation by defining the semantics of their definition language using automata. Yemanja is a model-based event correlation engine for multi-layer fault diagnosis [1]. The entity model represents devices or abstract components and encapsulates the entity behaviour. This correlation engine targets correlating low-level network events with high-level application performance alerts that are related to quality of service violations. These approaches focus on offering sophisticated facilities for implementing event correlation. It however remains a challenging research issue to provide abstract, implementation independent semantics to correlate events. More specifically there is little support for the representation of events transformation and filtering required within a middleware platform.

Recent work on publish/subscribe based computing systems has motivated the design of event correlation technologies using publish/subscribe architectures. The aim is to filter events and reduce communication overhead. June et al [15] define a formal semantics for a sequence-based event correlation framework. This framework has been incorporated into CORBA Component Model (CCM) and the event correlation has been integrated with the CCM event type system.

More recently, event correlation technologies have been considered as the core modules to build autonomic computing systems [17]. Beyond the basic functionality of reducing event traffic and communication overhead, event correlation is also used to diagnose symptoms and discover potential problems in order to achieve better results and eliminate problems. These tasks require event correlation to be tightly integrated with the control loop that implements adaptive behaviour including collecting data, analysing data, planning an action and executing the plan. The IBM Active Correlation Technology (ACT) [2] provides a rule builder tool that can take out of the box actions and event definitions as inputs and generate correlation rules in XML. However, the implementation of ACT remains an integrated part of IBM’s software solution for autonomic computing instead of a stand-alone tool.

Growing needs from various domains such as financial, telecommunications and the airline industry also require event correlation for matching event patterns that can occur from a large number of event sources. In particular, the event correlation function is considered in Event Driven Architectures (EDA) [13]. EDAs are complementary to Service Oriented
Architectures and event correlation is part of an EDA infrastructure. Therefore event correlation needs to be seamlessly integrated with middleware platforms supporting SOA such as Web services, an Enterprise Service Bus and messaging queues.

In a summary, event correlation is widely required in middleware-based distributed applications. However, little attention has been paid to the issue of designing a configurable and flexible event correlation architecture that balances quality requirements for both performance and flexibility.

Our focus in this paper is to design and implement an architecture that supports the event correlation requirements for adaptive control in middleware-based systems. This architecture is required to be deployed in distributed environment. We implement sequence-based correlation for events produced and consumed at the middleware level. This event correlation function is seamlessly integrated with an SOA-based distributed computing environment. The effectiveness of our architecture design is validated by applying the design to a real adaptive Java server application.

3. Event Correlation Architecture

We consider event correlation functionality in the context of adaptive middleware platforms, in particular, Java-based adaptive application servers. A fundamental design principle of augmenting application servers with adaptation is to separate the adaptive behaviour implementation from the application components. The adaptive behaviour is encapsulated into external control components [5] and plugged into the application servers. The event correlation architecture hence must follow this paradigm and be integrated with other control components through a generic event channel.

3.1 Reference Architecture for Adaptive Middleware-based Applications

Event correlation is one of the key components required to realize adaptive control. Our reference architecture establishes the context for event correlation and defines how event correlation elements should interact with other components.

The reference architecture in Figure 1 depicts the interaction among application components, control components and the underlying servers. The control components form a control loop implementing dynamic and adaptive behaviours to fulfil the adaptation requirements. The control components interact with the application server, monitor the runtime environment, analyse collected data, and change the application’s behaviour by modifying outputs or changing the middleware platform configuration.

As shown in Figure 1, connections among individual components are driven by events generated from monitoring and analysis. There are three types of connections: (1) connections between adaptive components and application components (see label 1) for collecting runtime information of the application; (2) connections between potentially distributed control components themselves (see label 2) for interchanging relative control events; and (3) connections between control components and the server (or middleware) (see label 3), which intercept requests and modify responses through the middleware.

Event correlation is implemented as a module called the event correlation engine (ECE). An ECE may take input events from any of the three connections discussed above, and generate an output as a new event that can be fed into other instances of the control components. For example, events generated from different monitoring components that contain runtime information may need to be correlated. As a result, the correlated events from
different monitors are the inputs for analyser components. The ECE encapsulates the filtering and transformation of events. Without an ECE, the code for event correlation has to be scattered around the implementation of the control components, which would incur unwelcome maintenance and extensibility issues.

Figure 1. A reference architecture of adaptive middleware-based Web applications

3.2 Configurable Event Correlation Engine

The design of the ECE architecture addresses several concerns, including:
1. The ECE should fit into the overall architecture of adaptive middleware-based applications as shown in Figure 1.
2. Event correlation should allow integration with Web services in a loosely coupled distributed architecture, especially for event dispatching and handling.
3. The overhead incurred by event correlation needs to be optimized.

In addition, the architecture should be able to support reliable event correlation so that events can be buffered when the arrival rate overwhelms the processing capacity of the ECE or there are failures in processing events. Moreover, the architecture of the ECE should be flexible and allow implementations to address either performance or reliability requirements.

The major components of the ECE are illustrated in Figure 2. Event producers and consumers are specialized components used for realizing adaptation logic, such as monitors and effectors. The ECE between event producers and consumers is the core component that correlates events at runtime when events flow through the ECE and events patterns are detected by rules specified as regular expressions. Rule specification will be discussed later in section 3.5.
The design of the internal structure of the ECE follows the proxy design pattern. Instead of directly providing access to the Event Correlator, we introduce an Event Proxy for accepting events from event producers and redirecting them to the Event Correlator. This design has two benefits. First, it improves the extensibility as the proxy shields the producers from changes to the correlation components. Second, this architecture supports the distribution of event correlation. The Event Proxy and the Event Correlator can be distributed on different nodes and connected by different types of middleware protocols such as RMI, messaging or Web services. The Event Correlator’s internal structure, the semantics of event correlation rules and implementation techniques are briefly introduced in the next section.

Events may arrive at the ECE at different rates from different sources, and there may be a delay between the time an event arrives and the time when it is actually processed. Consequently, when a large number of events arrive at the ECE, actions must be taken to avoid losing events or consuming excessive resources. Moreover, different events may have different priorities and QoS requirements for reliability and performance.

In order to create reliable and high performance architecture for correlating events, the connection between the Event Proxy and the Event Correlator in Figure 2 is implemented as a message queue using the Java Messaging Service (JMS) [11]. This solution provides reliable event transfer and holds unprocessed messages. It takes advantage of messaging middleware to simplify the management of events.

An event can be configured either as a reliable event or a high performance event. Only a reliable event is passed through JMS. A proxy component inside the engine accepts an event, reads the configuration of the event and decides whether to dispatch the event to the ECE directly or put it in the queue. The ECE proxy and message queue are transparent to the message producer and consumer components.

In this architecture, reliable event correlation depends on the ability of the JMS server to deliver and recover events. A method described by Chen et al. [4] can be followed to evaluate the reliability of a message oriented middleware; however its evaluation is out of the scope of this paper. The JMS server can be configured in persistent mode, so that each event that arrives at the JMS queue is persisted and hence recoverable.

3.3 ECE Lifecycle management

Event producers and consumers are not aware of the existence of the ECE. Events to be correlated are specified in a configuration file, and this configuration is interpreted by an adaptive engine, whose role is coordinating and managing all the control components, including the ECE. Details of this adaptive engine, known as ASF, are described in [7].
ASF has a bootstrap function that loads all control components when a server starts up or a new control components package is deployed. The engine reads a configuration file and obtains information on individual components and their connections. Figure 3 shows an example configuration for the ECE. When the ASF starts, it creates an instance of the ECE SimpleCorrelator. The ECE can then initialize the types of events as specified in CONSUMED_EVENT_LIST and PRODUCED_EVENT_LIST. The resulting correlated event ImageScaleSyncMsg is dispatched to the consumer component ImageScaleEffector at runtime using publish/subscribe communications.

```
# Declare an ECE
CORRELATOR_LIST=SimpleCorrelator
SimpleCorrelator.CLASS=au.com.nicta.correlator.SimpleCorrelator

# Specify SimpleCorrelator's attributes
SimpleCorrelator.CONSUMED_EVENT_LIST=InvokeProxyMsg,ImageScaleProxyMsg
SimpleCorrelator.PRODUCED_EVENT_LIST=ImageScaleSyncMsg
SimpleCorrelator.ImageScaleSyncMsg.CONSUMER_LIST=ImageScaleEffector
```

Figure 3 A sample configuration of ECE

3.4 The Internal Structure of Event Correlator

The internal structure of the event correlator is elaborated in Figure 4, which includes three major components, the event buffer, the filter and the transformer. The event buffer holds incoming events awaiting processing. The initial size of the buffer is configurable. The filter is associated with a set of expression rules denoting the event patterns to be detected. The semantics of the expression rules are described in the next section. The transformer encompasses code that represents the action to be taken. Normally the action involves tasks to transform original events into a new composite event and dispatch the composite to its destination. Splitting the event correlation functionality into a filter and a transformer allows the event filter to be integrated with more complex knowledge-based management of rules [17].

Figure 4 Internal structure of event correlator

3.5 Event Format
In complex situations, events can be produced from multiple sources in a heterogeneous environment. This requires the availability of contextual information that represents the event generation context. Ogle et al from IBM have proposed the Common Based Event (CBE) specification to define the structure of an event in common XML format [20]. CBE is used in the Eclipse Test and Performance Tools Platform (TTP) project and IBM’s Autonomic Computing Toolkit. In this paper, we adopt CBE as the event format of the ECE.

The structure of CBE has three main tuples, namely the component reporting a particular situation, the component affected by the situation, and the situation itself. In the CBE component model [8], components either reporting a situation or being affected by a situation are modelled as a ComponentIdentification class. A situation is represented by a Situation class. A situation can be further decomposed into several subclasses such as StartSituation, StopSituation, CreateSituation, FeatureSituation, ReportSituation, ConfigureSituation, DependencySituation, and so on. All these three tuples are encapsulated into the CommonBaseEvent class, which has the meta-data about an event, and methods to obtain data about an event. Instance of these classes can be serialized to and instantiated from XML streams.

The CBE defines data elements for several types of events including logging, tracing, management, and business events. All these data elements are defined in a consistent format, which allows these events to be correlated in an interoperable manner. The CBE defines properties for each event type and provides APIs to manipulate the event payload. Figure 5 demonstrates an example using the CBE API to create and configure an event. It demonstrates attaching customized information to the event through an ExtendedDataElement in ECE. For example in ECE, the QoS configuration of an event requires Reliability can be set as an ExtendedDataElement (see line 16 in Figure 5). The CBE instance can also be serialized and converted to a XML string.

```
1. //Create a typed EventFactoryHome instance
2. EventFactoryHome eventFactoryHome = new RuntimeEventXMLFileEventFactoryHomeImpl();
3. //Retrieve an EventFactory instance
4. EventFactory eventFactory = eventFactoryHome.getEventFactory("HyadesEMFCommonBaseEventv101Sample ");
5. //Create a new instance of a situation
6. Situation situation = eventFactory.createSituation();
7. situation.setCategoryName("ReportSituation");
8. situation.setSituationType(reportSituation);
9. //Create a new instance of a Common Base Event
10. CommonBaseEvent commonBaseEvent =
    eventFactory.createCommonBaseEvent();
11. commonBaseEvent.setSituation(situation);
12. commonBaseEvent.setSeverity(((short) (60)));
13. //Add some ExtendedDataElements
14. commonBaseEvent.addExtendedDataElement("CPUSensorValue", "0.65");
15. //Set the configuration for QoS (reliability or performance)
16. commonBaseEvent.addExtendedDataElement("QoS", "Reliability");
17. //Convert CBE into XML string
18. String eventString =
    EventFormatter.toCanonicalXMLString(commonBaseEvent,true);
```

Figure 5 An example of creating and configuring a CBE-based event

3.6 Semantics of Event Correlation Rules
Our current implementation supports sequence-based event correlation. We adopt the following annotations to express the basic operations:

- **Operator +** denotes that both Event A and Event B occur. There is no restriction of the order of A and B;
- **Operator |** denotes that either Event A or Event B occur;
- **Operator ;** denotes the order of the event. The event of the second operand follows the event of the first operand;
- **Operator *** denotes any events. For example, $A*;B$ means before Event B occurs there can be any event occurs following Event A.
- **Operator (n)** denotes the number of occurrence of an event. For example $A(5)$ means Event A occurs 5 times.
- **Operator [a]** denotes the attribute field that is used for correlating two events. For example, $A[id]+B[id]$ means Event A and Event B are correlated only when they have the same value of the attribute id.

These operators are used by application designers to configure the event sequence that needs to be correlated. The operators are then converted into Java regular expressions so that they can be processed by the ECE.

A combination of two operands and an operator is defined as a unit expression. An example of a unit expression can be $A+B$ or $A*B$. In order to provide a method for complex and nested expression, labels are used to denote unit expressions. A label contains a unit expression and the combination of n-level of labels will provide any possible complex expression. For example, the complex expression $((A+B);(C|D))*(A+C)$ can be configured using labels as the following:

<table>
<thead>
<tr>
<th>#Labels used for regular expressions</th>
</tr>
</thead>
<tbody>
<tr>
<td>NumberOfLabel=5</td>
</tr>
<tr>
<td>Label1=L1:L2*L5</td>
</tr>
<tr>
<td>Label2=L2:L3:L4</td>
</tr>
<tr>
<td>Label3=L3:A+B</td>
</tr>
<tr>
<td>Label4=L4:CID</td>
</tr>
<tr>
<td>Label5=L5:A+C</td>
</tr>
</tbody>
</table>

Table 1: Example of using labels

4. **An Event Correlation Service**

The component model of the ECE as described in the previous section forms the core of the event correlation architecture for a single server deployment. However, it has limitations in supporting the ECE deployment in a highly distributed, loosely coupled environment. A solution to this problem is to expose the ECE as a Web service, which we refer to as the Event Correlation Service (ECS). Event producers dispatch events to the ECS by sending Web service messages. Similarly the ECS dispatches newly composed events to the event consumers through a Web service invocation.

Deploying individual components in the ECE architecture as Web services requires exposing their implementation using a Web service interfaces. The core ECE implementation
is reused in this design. This is achieved by applying the wrapper design pattern, introducing a wrapper class for each component. The wrapper class implements the Web service interfaces, and is referred as the Web service wrapper in Figure 6.

With the ECS, it is important for all of the event producers to send events to the same instance of the Event Correlator so that sequence-based correlation can be performed correctly. The Event Proxy Web service wrapper is responsible for ensuring that all of the event producers can share the same instance of the Event Correlator.

In order to perform sequence-based correlation, it is critical that the ECS remembers information about the events which are dispatched to it. However, the default style for Web services (stateless Web service) does not support this. More precisely, stateless Web services do not keep state internally from one invocation to another. Consequently, in order to address this problem, we decided to implement the service wrapper using stateful Web service.

WS-Resource Framework (WSRF) [22] is a specification by OASIS that defines a generic and open framework for modeling and accessing stateful resources using Web services. WSRF is supported by a set of Web service standards, including WS-Resource, WS-ResourceProperties, and WS-Addressing. The major construct in WSRF is the WS-Resource. WSRF defines how a WS-Resource is composed of a Web service and a stateful resource. This separates the resource with state information from the service implementation. The Web services are still stateless, but their associated state and its lifecycle are managed by a resource. A resource can also be shared among multiple instances of the Web service. An abstract program model illustrating the interactions between the major components using WSRF is shown in Figure 7.

![Figure 6 The structure of ECS](image)

![Figure 7 WSRF programming model](image)
In WSRF, the state of a WS-Resource is modelled as a Web Service Resource Property (WSRP). It can be queried and modified via Web service message exchanges. The response message contains the endpoint reference of a WS-Resource, which is uniquely identified by its ID. The resource ID is actually an attribute of the ReferenceProperties in WS-Addressing (shown in Figure 8). From the point of view of the service requestor, the endpoint reference represents a pointer to the WS-Resource. Through this pointer, individual resources can be identified by a resourceID.

```
<wsa:EndpointReference>
  <wsa:Address>
    http://someOrg.com/aWebService
  </wsa:Address>
  <wsa:ReferenceProperties>
    <tns:resourceID> C </tns:resourceID>
  </wsa:ReferenceProperties>
</wsa:EndpointReference>
```

**Figure 8 Endpoint reference containing a stateful resource identifier**

In sequence-based event correlation, the order of events is session-based state information and this order must be maintained between individual invocations. This state information can be modelled as stateful resources in the WSRF. In WSRF, the events used to identify a sequence is the resource of the services, and is implemented as a resource property. We have built this architecture using the Globus Toolkit (GT4) Web service core container [18], which directly supports the WSRF programming model depicted in Figure 7.

The ECS architecture adopts CBE as the interoperable event format to support communication among event producers, the ECE and event consumers. Events are exchanged through the dispatchEvent and handleEvent methods supported by event producers and consumers respectively. The manipulation of CBE components to access event data is illustrated inside the implementation of both interfaces in Figure 9. Note that the method signatures of dispatchEvent and handleEvent are annotated with @GridMethod. Each annotated method becomes a stateful Web service interface. This annotation is from the Grid Development Tools (GDT), which supports the development of stateful Web services by automatically generating stubs and WSDLs that can be deployed to the GT4 container.

The default implementation of the ECS is built on the JDK v1.5. The processing of correlation rules utilizes the Java regular expressions package. We use JBoss JMS as the default message queue in ECE, but it can be replaced by other Java-based message queue middleware. The CBE Java library is available from IBM’s Emerging Technology Toolkit (ETTK) [8]. We use the CBE API to compose events and access event attributes. The stateful Web service is implemented using GT4 Web service core container [18].
5. Case Study

We illustrate the usage of event correlation service by a case study. This case study integrates the event correlation architecture with an adaptive image server application. This adaptive J2EE application has been implemented using the Adaptive Server Framework (ASF) developed in [7]. The ECE is utilized to correlate events from multiple control components.

5.1 Adaptive Image Server Application

The functional implementation of the image retrieval application is illustrated in Figure 10. A client sends a request to the application server for a specified image along with a minimum and maximum resolution for the image. By default, the maximum resolution is 1.0, which means the original image is returned without scaling. A J2EE application server hosts the image processing application using EJBs, which sends a request to retrieve the image from a database, where images are stored as BLOBs (Binary Large Objects).
1. request a image with file name, minimal resolution, maximum resolution
2. retrieve the image saved in the BLOB field
3. scale the image
4. return the scaled image back to client

**Figure 10 Image Retrieval Use Case**

The business goal of this application is to improve the throughput and reduce the response time of the image server. Without adaptation, the image server just returns an image with its original resolution and returns it to the client. Given that clients request a minimum and a maximum resolution for an image, the application is free to choose the resolution and image quality it delivers in order to optimize its performance. Scaling an image takes CPU time, and the image size affects the image transport time. Hence the application can adaptively select the image resolution based on a simple analysis model to trade-off the scaling computation cost and reduced network latency.

We have studied how image scaling with different levels of resolution and quality can affect performance. This has been done by taking empirical measurements of scaling over 100 images with the image size range from less than 1KB to over 2MB. We summarized the relationship between the scaled image size, scaling time, and image quality as follows:

The scaled image size depends on both resolution and quality, while the image scaling time is mostly affected by the resolution. The effect of the image quality on the image scaling time is not significant. The higher the resolution or quality, the larger the final image size is. In addition, the image scaling takes longer as the resolution level increases.

From the empirical observation and analysis, we can infer that image scaling time affects the application’s response time, and the scaled image size determines the delay of transferring images over the network for a given network speed. The goal of the adaptation is based on the workload of the server and the network connection speed. The server should adaptively return images at different levels of resolution and quality that can both meet the client requirements for the images, and optimize the performance of the application server under peak load.

5.2 Event Correlation Design and Implementation

The implementation of the adaptation trades off image resolution and quality against image processing time to reduce the transport overhead over the network. Based on the above analysis, we can design and implement the following components to create a control loop as shown in Figure 11 to determine how an image should be scaled:

1. *ImageScaleEngine* is the engine component responsible for bootstrapping and managing all the other control components.
2. *BandwidthSensor* intercepts a client request and detects the client connection network speed and the request arrival rate. It forms an event with these details and sends it to the
monitor. It also assigns a unique id to each invocation to differentiate different clients requesting the same image.

3. **ImageScaleMonitor** takes the **BandwidthSensor** input event, attaches the CPU usage information to the event and further sends it to the analyser.

4. **ImageScaleAnalyzer** implements the analysis model that represents the relations of metrics described above.

5. **ImageScaleEffector** receives an event from the analyser indicating the optimal resolution and quality settings to scale an image. This effector intercepts the return method of the invocation and replaces the return result with the image scaled according to the quality and resolution analysed. Figure 11 shows that the effector’s actions depend on two flows of events:

- the flow of the request invocation passing through application components, where the interception of the request and the replacement of the reply take place;
- the flow of the control events going through the monitor and the analyser, which contains the action content of the effector.

![Figure 11 The conceptual control loop for scaling an image](image)

As highlighted by the bold text in Figure 11, without the ECS service, these two flows of events have to be correlated inside the **ImageScaleEffector**. Obviously, this solution couples the process of event correlation with the control logic implementation of the effector. Moreover, synchronizing events inside the effector introduces computation overheads and delays the processing of the effector. In order to solve this problem, we introduce the event correlation service into the architecture in Figure 11. Recall that the **BandwidthSensor** assigns a unique id to each invocation from the clients, and this id is kept while the request passes through the application components. Therefore, we can replace the ECS in front of **ImageScaleEffector** to synchronize the invocation of the application components and the events through the analyzer using their event id.

In a single server environment, the deployment of ECE only performs event correlation for a single server. In order to illustrate the use of ECS in a distributed architecture, we extended the deployment of the image server application into a clustered server environment. We configured two Java application servers in one cluster with each server deployed on an identical machine. Each server in the cluster has instances of adaptive control components, including **ImageScaleEngine**, **BandwidthSensor**, **ImageScaleMonitor**, **ImageScaleAnalyzer** and **ImageScaleEffector**. When clients send requests to the cluster, the requests are automatically allocated to one server node in the cluster.
As shown in Figure 12, the ECS is deployed as a Web service in a separate stateful Web service container hosted by a Tomcat server. The endpoint information of the ECS is specified in the configuration files for the control components (in the same manner as illustrated in Figure 3). When each server starts, it deploys the adaptive control components and initializes their instances. The endpoint of the ECS can be set in the control components, which then interact with the ECS to either send or receive events. In this clustered environment the ECS correlates two flows of events, namely the request invocations and control events for each individual server. In addition to the event properties required for the adaptive control, an attribute indicating the server node identity is attached to the event when the request is accepted by a server.

We implemented the case study using JBoss Server 4.0.1 and JRE 1.5 on Windows XP. Two JBoss server instances are started in one cluster, with each server running on a separate machine. The adaptive components for this case study leverage ASF’s component model [7]. A Java client application simulates the image server workload by starting a number of threads and simultaneously sending requests to the server. Each client randomly picks a file name as its parameter for the request to the server, and the server component (a session EJB) returns the corresponding image as a byte stream. Our test data has images stored in an Oracle 9i database as BLOBs, with sizes from 800 bytes to 2.3MB. Our lab environment supports a shared 100Mbps Ethernet, with the throughput of a network connection approximately 960Kbps. The client, application servers and database machines are identical. All are workstations with Dual Intel Xeon 3.00GHz CPUs and with 3G RAM, running Windows XP.
6. Evaluation

There are two important quality attributes for this event correlation service, namely performance and flexibility. We have already demonstrated the flexibility of the event correlation architecture in both the component based single server and the service oriented server cluster in section 5.2. In this section, we evaluate the performance of our event correlation architecture implementation in terms of the overhead incurred by event correlation. The overhead includes the time spent on event correlation itself, transferring messages through the JBoss JMS and stateful Web services. We further demonstrate the performance benefits of using event correlation by comparing the response time of the adaptive image server application under three scenarios, namely without adaptation, adaptation without event correlation (synchronizing events inside the effector) and adaptation with event correlation.

6.1 Overhead of Event Correlation Engine

The ECE provides application designers with architectural alternatives to trade-off between performance and reliability. Reliability is achieved by introducing the extra overhead of a message queue. In order to quantify the performance overhead of this architecture, we insert timing probes into the ECE components and attach timestamps to the JMS messages that wrap event data to measure the delay introduced by the ECE and the JMS message queue.

The results demonstrate that the overhead introduced by ECE is negligible when the workload is light. The overhead is approximately 1 millisecond for an arrival rate of 10 messages per second. We examine the overhead of the JMS message queue under the condition that the ECE is configured to be reliable. Figure 13 shows the delay time of the JBoss JMS server on our test bed, which is around 4 milliseconds for an arrival rate of 10 messages per second, and it scales well until the arrival rate reaches 100 messages per second. Beyond that, the delay time of the JMS increases non-linearly. Based on the measured JMS performance and the intensity of event correlation, the designer can decide the QoS configuration that best suits the performance requirements.

![Figure 13 Performance and scalability of JBoss JMS on the test bed](image-url)
6.2 Overhead of Stateful Web Services

The loosely coupled nature of Web services make it nontrivial to measure the delay of Web service invocation, especially the overhead involved in the state management. We therefore propose a simple estimation technique by decomposing the delay into the overhead of Web service invocation, event correlation processing, and the time spent on the resource state management. The events flow in the case study is illustrated in Figure 14. In the diagram:

- \( T_{\text{total}} \): the interval between the time when the event leaves the event producer component and the time when the event arrives at another ASF component after being correlated.
- \( T_{\text{ecs}} \): is the interval between the time when the event arrives at the ECS and the time when the event leaves the ECS, which includes the delay of the stateful resource management in Web services.
- \( T_{\text{ws}} \): the time it takes for the event to reach the ECS after leaving the event producer component or the time it takes for the event to reach the event consumer component after leaving the ECS. Since both of these intervals are Web service communication, it is assumed that they are equal.

The relationship between those intervals is that \( T_{\text{total}} = T_{\text{ecs}} + 2 \times T_{\text{ws}} \).

In order to evaluate the overhead of the case study, \( T_{\text{total}} \) and \( T_{\text{ecs}} \) are measured in different scenarios in terms of the number of clients sending requests simultaneously. The number of clients also represents the workload of the Event Correlation Service. For example, 16 clients mean 16 events arrive at the ECS simultaneously. The interval between event arrivals is one second. We can then derive an estimate for \( T_{\text{ws}} \), and the results are illustrated in Table 2. These indicate that as the arrival rates increases up to 16 clients, the overhead of event correlation service increases significantly.

\( T_{\text{ecs}} \) includes two factors, both the performance overhead of the event correlation mechanism and the delay of the stateful resource management, which are difficult to measure separately. Here we estimate the performance overhead caused by each factor. The evaluation in section 6.1 shows the ECE architecture without Web services is scalable to an arrival rate of 100 events per second. However, \( T_{\text{ecs}} \) increases significantly when there are 16 clients. It appears that the associated Web services infrastructure needed to create stateful services compromises performance and scalability. Further optimization of the stateful Web service implementation can help to improve the performance and scalability of our ECS architecture.
For performance sensitive distributed adaptation scenarios, the overhead incurred by the stateful Web service needs to be taken into account especially when the system has high demands for event correlation. The architecture remains useful for adaptation scenarios that need to seamlessly correlate events from multiple distributed sources with a moderate performance demand.

<table>
<thead>
<tr>
<th>Scenarios</th>
<th>T total (ms)</th>
<th>T ecs (ms)</th>
<th>T ws (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 client</td>
<td>77.78</td>
<td>39.48</td>
<td>19.15</td>
</tr>
<tr>
<td>2 clients</td>
<td>83.75</td>
<td>45.42</td>
<td>19.165</td>
</tr>
<tr>
<td>4 clients</td>
<td>84.50</td>
<td>49.60</td>
<td>17.45</td>
</tr>
<tr>
<td>8 clients</td>
<td>133.51</td>
<td>95.03</td>
<td>19.24</td>
</tr>
<tr>
<td>16 clients</td>
<td>212.97</td>
<td>165.77</td>
<td>23.6</td>
</tr>
</tbody>
</table>

Table 2 Overhead value

### 6.3 Performance Improvement Using Event Correlation

We further compare the performance of the adaptive image server for three scenarios: 1) with adaptation enabled but without using ECS; 2) with a daptation enabled and using ECS; 3) without any adaptation. We also compare the performance of two QoS configurations using ECE, with priority set on either on reliability or performance.

We categorize the workload according to the image size into four groups, namely less than 10KB, 10KB-100KB, 100KB-500KB, and 500KB-2MB. The client requests for images are evenly distributed among the four groups. To evaluate the performance under a varying request load, we generate a workload by periodically changing the number of emulated clients. The number of concurrent clients is increased in a step function of ten every two minutes from a starting number of five until a maximum of sixty-five. After that, at two minute intervals, the number of clients is decreased by 10 until 35 remain. The intent of this workload is to mimic sustained bursts of increasing workload against a backdrop of moderate activity. Each step in the workload produces a different plateau of workload level. Therefore the workload stimulates the engine to adapt to CPU usage on and below the peak level. The entire test lasts for twenty minutes. The same workload is applied to the server application executing with and without adaptation. The configuration of JBoss servers is exactly the same for the three tests.

![Response Time Distribution](image_url)

**Figure 15.** Response time distribution with adaptive behaviour disabled
Figure 16. Response time distribution with adaptive behaviour enabled (without event correlation)

Figure 17. Response time distribution with adaptive behaviour enabled (with reliable event correlation)

Figure 18 Response time distribution with adaptive behaviour enabled (with high performance event correlation)

Figure 15 to Figure 18 show the client response time distribution for these three tests. Without adaptation as shown in Figure 15, 90% of response times are within 36 seconds and
the overall throughput is 187 transactions per minute. Figure 16 shows that with adaptation executing but not using event correlation, 90% of response times are within 23 seconds and the overall throughput is 240 transactions per minute. The overall performance is further improved by using event correlation to correlate events. In Figure 17, the events are all configured as reliable messages to give a worst-case performance evaluation of the ECE. The overall throughput is 252 transactions per minute and 90% of response times are within 18 seconds. In Figure 18, the events are then configured as high performance messages to achieve the best performance of the ECE. The overall thought is 268 transactions per minute and 90% of response times are within 15 seconds. It is clearly shown in Figure 15 to Figure 18 that the overall performance in term of the response time is significantly improved using event correlation. The maximum response time is reduced from approximately 27 seconds without using the ECS to less than 20 seconds using high performance event correlation.

7. Discussion and Reflection

7.1 Integration with other types of middleware

In this paper, we demonstrate our event correlation service architecture in an example adaptive image server application executing on distributed JBoss servers. We adopt the CBE as a generic event model for containing event information. Therefore the architecture is independent of the middleware platform. We have successfully integrated our event architectures with JBoss servers, the GT4 Web service container and Tomcat servers, and we believe it can be directly deployed (or with minor configuration effort) on other middleware servers.

7.2 Optimization of performance overhead

The performance overhead introduced by the stateful Web services and the extensibility they bring to the ECS architecture are trade-offs in a distributed environment. Due to the overhead, the benefit of our event correlation architecture in a server cluster is less impressive than the single server scenario in terms of the throughput and response time. However, in other scenarios rather than performance critical ones such as problem diagnosis, the event correlation service can still benefit the overall architecture by achieving good separation of concerns. This is because the ECS encapsulates the complex event correlation computing and provides the Web service invocation. In order to reduce the performance overhead, it requires middleware vendors optimize the implementation of stateful Web service standards. The performance of stateful Web services is out of the scope of this paper. Ongoing research work is covered by Humphrey, M. et al [8]. It remains our future work to compare several stateful Web service containers including IBM’s Autonomic Computing Toolkit and select a container with better performance.

7.3 Enhancement of event correlation services

Correlation techniques are well covered in the literature. The challenging issue is to integrate these techniques with an effective event correlation architecture. In this paper, we demonstrate our approach to the design and implementation of a service-based architecture for event correlation. Our approach is driven by the need to integrate event correlation with an existing framework for developing adaptive middleware-based systems. Our experience
shows that middleware technologies, such as JMS and Web services can help to achieve flexibility and reliability in adaptive server applications.

8. Conclusions

In this paper we describe an event correlation service for distributed middleware-based applications. We address the core architecture and the deployment of this ECS in a loosely coupled distributed environment by utilizing the CBE format and stateful Web service technologies. The internal structure of the event correlation service enables the configuration of an event to tune reliability and performance. This is achieved by introducing proxy components that transparently interact with the underlying queuing technology. Our empirical evaluation on a test bed demonstrates the performance characteristics of this architecture. The result shows that the core components of the ECS produce good performance and scalability. The major overhead is introduced when the stateful Web services are applied to handle events generated in a distributed environment.

The current version of the ECS only supports sequence-based event correlation. Our ongoing work is to extend the ECS with other event correlation patterns, and to evaluate their impact on performance and scalability of the overall architecture. Another limitation of this architecture is that the ECS is not integrated with advanced knowledge-based rules and policies. With this established based of event correlation, we are planning to further extend its capabilities in this direction.

9. References


Appendix: Revision Report

The authors pay close attention to the review comments. The comments are addressed carefully in this revision. Some comments share the common opinion. We first summarize the key points of these comments and provide descriptions on how these points are addressed. We then elaborate some specific points.

1. Presentation and English
   Grammar mistakes and typos are fixed by proof-reading in this revision. The structure of this paper is also improved. We rewrite the abstract and introduction. We reorganize the structure of section 3, 4 and 5. The writings are also modified to provide a more clear and concise descriptions.

2. Related Work
   One reviewer pointed out an important reference missed from this paper. This is fixed by reference paper in [23]. More discussions of the related work is provided in paragraph 2.3,4 on page 3.

   It is pointed out by two reviewers that it is not clear what the new contribution of this paper is compared to the related work. This is now addressed in the last paragraph of section 2 on page 4.

3. Case Study
   It is pointed out by two reviewers that the presentation of the case study is not clear. The description of the server cluster scenario is not tightly linked to the description of the case study. We improve the structure and writings of section 5. Please see the new section 5.1 and 5.2.

4. Trade-off of the architecture.
   The explanation of the architectural trade-off was not explicit. This is partially addressed in the revised section of 5.1 and 5.2. A more explicit summary is addressed in the beginning of the section 6 on page 16.

5. “The performance overhead of the event correlation mechanism (T_ecs) also depends on the complexity of the filtering / correlation rules. Unfortunately, the complicity of these rules and its influence on the performance is not discussed. (i fthere is only a minor influence, this should be pointed out explicitly as a benefit of your approach.)”

   This is now addressed in paragraph 2 of section 6 on page 18.

6. “Please discuss not only the 90%-quantile of the results. From the figures given, it seems that the ranking of the different approaches might be different for lower quantiles.”

   This is now addressed in the last paragraph of section 6 on page 20.
7. “One particular aspect I am missing in this paper is event lifecycle: when are events considered to be consumed?” This is now explicitly addressed in the newly added section 3.3 on page 7. The case study description in section 5.1 and 5.2 also covers this point now.