Pluggable Java EE-based Architecture for Efficient Coupling of Event-based Applications

Samuel Dauwe, Bruno Van Den Bossche, Filip De Turck, Bart Dhoedt, Piet Demeester
Department of Information Technology, Ghent University - IBBT, Gent, Belgium

Abstract—In many enterprises, business functions are spreaded across multiple applications. In order to provide efficient, reliable and secure data exchange between applications, these components need to be integrated. In this context, component based technologies have become very popular. Component architectures such as Java EE facilitate rapid and flexible application development. Nowadays, messaging system are frequently used to enable high-speed, asynchronous, program-to-program messaging. In spite of the many existing asynchronous communication solutions, there is a need for an efficient generic messaging model.

In this article we present the architecture of a pluggable messaging system targeted at enterprise applications. This model was used to implement a fully functional prototype using Java EE specific technologies. To evaluate our messaging system, performance tests were done. Results from this evaluation show that the system is capable of efficient asynchronous communication.

Keywords: Java EE, event based, software architecture

1. Introduction

The growth of the internet and the increased attention to object-oriented programming has resulted in a rapid adoption of distributed component architectures. These architectures provide the foundation and standards for invoking objects within one application from another application independent of their location.

Traditional applications use synchronous function calls; for example: one method calling another method, or one procedure invoking another remotely through a remote procedure call (such as RMI [1], CORBA [2] and DCOM [3]). Synchronous calls imply that the calling process blocks while the other process is executing a function and can only proceed when control is returned. In this model all systems have to be online and functioning flawlessly, resulting in tight semantic coupling. This point-to-point and synchronous communication approach is very suitable in rigid and static applications. However, with the increasingly dynamic nature of large-scale complex systems today, there is clearly a need for a middleware providing asynchronous messaging. We illustrate the importance of the asynchronous messaging principle by presenting a message-driven application. Consider an Air Traffic Control System, used for monitoring and controlling arriving and departing airplanes. First of all there are various sources of information like airplane transponders providing location information, weather stations and traffic updates from neighboring airports. This large amount of data has to be processed and presented in real-time. On the basis of this information, decisions can be made, which have to be communicated to the airplanes. It is clear this kind of system is difficult to realize using synchronous components. Firstly, there is the dynamic nature of the application. Moreover, communication between the varying number of components should be reliable. When data is packaged in independent messages or events, a store and forward approach can provide reliability. A sender can simply send the message to the system and continue with other processes, resulting in a asynchronous communication style.

These days, distributed multi-tiered application models like Java EE [4] and Microsoft.NET [5] are very popular and commonly used in the context of enterprise applications. In the Java Enterprise Edition there is the Java Message Service (JMS) [6] that can act as a communication channel. However, as event model it is not efficient enough, especially for heavily event-driven enterprise applications. First of all JMS is centered around the paradigm that each message is guaranteed to be delivered once and only once. This focus on reliability has consequences on the performance and the introduced overhead when creating messages [7]. Furthermore, all message channels must be known before deployment which makes it not very flexible. Although dynamic channel creating is possible, it is difficult to implement. When we consider technologies similar to Java EE, we notice similar disadvantages.

This paper focuses on the development and implementation of a framework that acts as a generic, event model between loosely coupled enterprise applications achieving efficient communication. Firstly, we give an overview of commonly used Java based application server environments usable to create event-oriented applications (Section 3). Next, we present the architecture of our messaging system (Section 4). We used this architecture to implemented a fully functional prototype using Java EE specific technologies as described in Section 5. An evaluation was done, investigation the most common performance metrics (Section 6). Finally, in Section 7 future work is stated an our conclusions are presented in Section 8.
2. Related Work

Traditionally asynchronous behavior in Java EE is realized using JMS. A number of design patterns make actively use of this asynchronous behavior to simulate threaded behavior to execute multiple tasks in parallel. By using the Splitter and Aggregator pattern [8] this can be achieved in the context of Java EE application servers. Using the event based approach proposed in this paper, the implementation of this pattern can be made highly efficient, compared to a the traditional JMS-based implementation. The same can be applied to the Service Activator pattern [9] to allow business services to be invoked asynchronously.

A typical use of the asynchronous invocation of business services is in the web-tier of enterprise applications making use of AJAX [10]. AJAX or Asynchronous JavaScript and XML, is a group of interrelated web development techniques used to create rich Internet applications. This allows web applications to retrieve data from the server asynchronously in the background without interfering with the display and behavior of the existing page. For example a servlet offers an initial version of the web page and the results of long running tasks in the back end application server are fetched using Asynchronous JavaScript.

An overview of the use and advantages of asynchronous communication for complex e-commerce applications is given in [11]. This allows to increase the throughput of the application significantly as shown in [12]. Traditionally JMS is used for these optimizations, however, using the low overhead event-based communication proposed in this paper allows to further increase the performance.

An Enterprise Service Bus (ESB) is an architectural pattern which defines a communication infrastructure between services. The main role of an Enterprise Service Bus (ESB) is to serve as a communication bus accepting a variety of input message formats and to transform these messages to different output formats and thus providing a transparent communication interface. A functional overview of commercial ESBs is presented in [13] and [14] evaluates the performance of four both commercial and non-commercial ESB implementations.

3. Java-Based Application Execution Frameworks

The Java Enterprise Edition (Java EE) [4], has emerged as a leading component architecture for applications requiring agility, scalability and interoperability. In this section we give an overview of existing Java-based application execution frameworks which are usable for the creating of event-oriented applications.

3.1 Java EE

The Java EE platform is essentially a Java application server environment that provides a set of Java extension APIs to build applications and a runtime infrastructure for hosting applications. Business applications are deployed and run in a container that manages application components and provides transparently access to Java EE services like transaction management, security checks and resource pooling.

The application container hosts Java EE applications composed of Enterprise Java Beans (EJBs). There are three type of EJBs: Session Beans which contain the business logic, Entity Beans which represent data and Message-Driven Beans which allow for asynchronous communication. Additionally a web-container is provided which enables the hosting of HTTP-based services. Figure 1 gives an overview of the Java EE logical architecture.

The typical way of communicating with a Java EE Application server is through synchronous method calls and the same goes for the inter component communication. Clients can interact directly with application components using a Java EE application client which performs Remote Method Invocations. For asynchronous communication, the Java Message Service (JMS) can be used in combination with Message-Driven Beans (MDB). JMS allows to configure communication channels called queues (one-to-one communication) or topics (one-to-many communication), MDBs consume and process the incoming messages on those channels. However, the capabilities of the JMS system are somewhat limited as the queues and topics are statically configured at deploy-time and no other efficient routing mechanisms are available in the framework.

Apart from this there is also support for deploying Java EE Connector Architecture Resource Adapters (RA) [15] into the application server. Deployable JCA components are called resource adapters. The Java EE Connector Architecture (JCA) defines a standard architecture for connecting the Java EE platform to a heterogeneous Enterprise Information System (EIS). An EIS provides the information infrastructure for an enterprise, examples EIS are ERP (Enterprise Resource Planning), database systems and legacy applications. By defining a set of scalable, secure, and transactional mechanisms, JCA enables the integration of an EIS with an application server and enterprise applications. The resource adapter plugs into an application server and provides connectivity between the EIS, the application server, and the enterprise application. The Java EE Connector Architecture was originally designed for implementing RAs to interface with existing (legacy) systems, but it can be used in a more general context to extend the Java EE application server with extra protocol stacks. For example, the JMS stack typically is implemented as a RA as well.

3.2 JSLEE

The Java APIs for Intelligent Networks (JAIN) [16] aims for an enabling set of Java APIs to develop and deploy service-driven network applications. The JAIN Service Logic Execution Environment (JSLEE) specification [17] provides
Fig. 1: Presentation of the Java EE logical architecture.

an application server targeted at Java communications applications demanding a high throughput, low latency event processing environment. Applications are composed of Service Building Blocks (SBBs) which are the equivalent of the Java EE EJBs. One of the key features of the JAIN SLEE application container is the use of asynchronous communication. Internally almost all communication in the SLEE happens by using events. The idea behind the event based communication between SBBs is that every SBB performs its own task and then hands the result off to the next SBB in line. One could compare this to an assembly line in a factory.

The internal routing is completely taken care of by the application server. A key component in the routing of events is the Activity Context which manages the links between logically connected SBB entities. When an initial event (for example the first message in a message flow) enters the SLEE an Activity Context is created. The SBBs processing this event will be attached to this Activity Context and all following events which logically belong together (e.g. all events related to the same message flow) will be fired on the same Activity Context and processed by the same SBB entities. This is important as the SBB entities processing messages from a single message flow use this Activity Context to share data. This is comparable to the use of Sessions in the context of Java EE.

Communication with the outside world happens through Resource Adaptors which are similar to RAs in Java EE. An RA is a component which can be plugged into the JAIN SLEE application server. This component then functions as a pass through for external events to the SBBs and the outside world and vice versa. Typically such an RA is used enable the JAIN SLEE to communicate using a certain protocol stack. Using this mechanism allows the JAIN SLEE application server to be protocol agnostic, i.e. it offers all required functionality for a generic event based application server and can be extended to support any protocol using RAs.

4. Architecture

In section 3 we gave a brief overview of JSLEE and Java EE. Although the two technologies are complementary, they are targeted at different types of applications. Because JSLEE is a specialized component model for event driven applications, it lacks some concepts which are present in Java EE. First of all, enterprise applications are more database access intensive. As a consequence, Java EE has better support for back-end systems and transactions, which is also reflected in the EJB specification. When we want to create event-driven enterprise applications, there is a need for a pluggable event model for Java EE application servers.

In this section we present the designed architecture of our messaging system aimed at enterprise applications, as shown in Figure 2. This event channel provides an asynchronous messaging service between the different application component which can reside in the same container or distributed on different machines. It consists of a set of modules interacting together and implementing several functional and non-functional properties like routing, reliability and ordering of message transfers. The requirements of the messaging system can be summarized as followed:

4.1 Requirements

- **Publish/Subscribe** Components should be able to register themselves with the messaging system by stating the event type they are interested in. Received events matching this interest will only be forwarded to those subscribed listeners. Each component should be able to attach and detach at any moment.
- **Asynchronous Communication** The sender should not have to wait for the receiver to receive and process the message; it should not even have to wait for the messaging system to deliver the message. This allows to run applications at maximum throughput and avoid unnecessary waiting time.
- **Mediation** The messaging system acts as a mediator between all of the programs that can send and receive messages. If an application becomes disconnected from the others, it needs only reconnect to the messaging system. The event channel should also provide access to shared resources such as a datastore.

In the remainder of this section we give an overview of each component in the presented architecture.

4.2 Component Description

**Application Component** An application component can be a server-side component, such as a Message-Driven bean which allows asynchronous invocation. Application logic is divided into components according to their function. When distributed components want to cooperate they have to exchange messages. Typically, each component should have the possibility to send or receive messages using the messaging system.

**Connection** The connection provides connectivity to the resource adapter. Components will first setup a connection with the event channel through which they can asynchronously send and receive messages. Beside sending
events there will also be the possibility to store session related data in a datastore.

**Event Handler** This component acts as an interface to the underlying network protocol. When messages are received from components residing in the local or remote container they often have to be translated to Java objects which the messaging system can handle.

**Event Processor** This module contains the actual logic of the resource adapter. Events received from the application components come together in the central buffer. Next, the event header of each message is analyzed by the event processor to determine the destination. The event processor drops events silently whenever there is a problem with the header, like unknown destination or wrong format. Because there is loosely coupling between sender and receiver, message producers are not notified when such a drop occurs.

**Logical Bus** According to the publish/subscribe paradigm, events are delivered to listeners according to their event type. Messages belonging to the same class should be treated the same, for example routed to the same subscribers. Therefore, we have chosen to introduce logical buses. These components will handle the events which have the same event type. Using this approach it is easier to provide a desired quality of service (QoS), another advantage is the fact that these buses operate in parallel increasing the overall performance. A logical bus has the following responsibilities:

- Filtering: Each logical bus is responsible for handling events which have the same event type. To provide quality of service there should be a filtering phase. This process will treat events differently according to extra information contained in the event header. For example, high priority messages will be handled first and placed in front of the queue in the next phase.

- Buffering: The publish/subscribe principle gives application components the opportunity to register and deregister themselves dynamically. Events received for a logical bus with no associated listeners should be buffered to ensure delivery. This buffer also gives the opportunity to deal with high loads.

- Routing: The dispatcher is responsible for the delivery of messages to the appropriate event endpoint. Routing is done by passing the event to the local or remote event endpoint proxy. The proxy passes the message to the endpoint, hiding, using the underlying communication protocol.

**Storage Adapter** Application components can use the connection to send session related data to the Storage Adapter. This adapter will generate an unique session identifier and handles data retrieval process.

**Datastore** Encapsulating large amounts of data in events should be avoided because they will decrease the overall performance of the event system. Furthermore, real-time communication is not always necessary for applications. In this case it should be possible for application components to store session related data and generate an event notifying the information is available. Next, other components can retrieved the data from the datastore, and continue their operations. With this approach, event sizes are kept small in size, furthermore when another datastore is required the only thing that have to be modified is the interface to the event model.

### 4.3 Component Interaction

To illustrate how components in the architecture interact we briefly describe a scenario when an event is exchanged. Figure 4 shows the sequence diagram of this internal communication scenario. The application sends an event using the resource adapter connection, which will be intercepted by the event handler. There is a handler for each connection, this component also contains protocol specific logic (e.g. creating Java objects from a bytestream). Next, the event is placed in a buffer where it remains until the event processor is able to process it. In a first step, the event processor analyzes the event header to determine if it is registration or deregistration or a normal message. According to the event type, logical buses are created and given to the matching bus. Finally, the bus will be responsible for the filtering, buffering and routing to the event endpoints. The event is delivered to the receiving application component by asynchronously invoking an interface method.
5. Implementation Details

The Java EE Connector Architecture (JCA) defines a standard architecture for connecting the Java EE platform to a heterogeneous Enterprise Information System (EIS). Deployable JCA components are called resource adapters and plug into an application server and provides connectivity between the EIS, the application server, and the enterprise application. Nowadays, the Java EE Connector Architecture is used in a more general context for example to extend the Java EE application server with protocol stacks. Because JCA makes it possible to develop a pluggable communication module, it was chosen as implementation technology for our messaging system. To achieve a standard system-level pluggability, the connector architecture defines a set of system-level contracts which are the following:

Connection management contract enables an application server to pool connections to an underlying EIS, and enables application components to connect to an EIS.

Transaction management contract enables an application server to use a transaction manager.

Security contract provides support for a secure application environment that reduces security threats.

Lifecycle management contract provides a mechanism for the application server to bootstrap a resource adapter instance during its deployment or application server startup and to notify the resource adapter instance during its undeployment or an orderly shutdown of the application server.

Work management contract allows a resource adapter to submit thread instances for execution.

Transaction inflow contract that allows a resource adapter to propagate an imported transaction to an application server.

Message inflow contract that allows a resource adapter to asynchronously deliver messages to message endpoints residing in the application server.

The Common Client Interface (CCI) defines a API for accessing EISs, used by application components. The connector architecture recommends, that a resource adapter support CCI as the client API. A client API specific to the type of a resource adapter and its underlying EIS could also be used instead.

A Resource Adapter allows bidirectional connectivity between a J2EE application and an enterprise information system (EIS). Unfortunately, the Connector Architecture does not specify the way of communication between components residing in different containers. With the Java Java 2 Standard Edition 1.4 it is possible to create non-blocking sockets. Previously, a thread had to be started for each connection to deal with multiple socket connections. This approach gave issues such as operating system limits, deadlocks, or thread

Fig. 4: Sequence diagram giving an overview of the interaction between the messaging system components when sending an event. The sending and receiving application components reside in the same container.

Fig. 5: Overview of the different contracts between the logical entities in the Java EE Connector Architecture.
safety violations. With Java NIO [18] one can use selectors to manage multiple simultaneous socket connections on a single thread, which makes writing scalable, portable socket applications simpler. We chose to use the Java NIO technology because it allows to create nonblocking sockets, resulting asynchronous high-performance communication between containers.

6. Evaluation Results

In this section we evaluate the capacity of our developed messaging system. The first performance metric we investigate is the latency, which is defined as the worst-case time taken to deliver messages from the sender to the receiver. This metric gives an idea of how efficiently the event channel can route messages. In our test scenario, we measure the latency for events with changing event size. Determining the introduced delay in a distributed environment is difficult, because all the systems clocks need to be synchronized precisely. Therefore, we run both the sender and receiver on the same machine so that the latency can be measured using the same clock. This approach has the disadvantage that communication components and messaging system all run on the same machine. Therefore, the CPU demand of the test components influence the messaging performance.

Figure 6 shows the used test setup. The sending component encapsulates the current timestamp ($T_s$) in an event. When the receiving component gets the message, it retrieves the current time ($T_r$). The messaging latency is then calculated using the formula: $\text{Latency} = T_r - T_s$

The performance evaluation of the event channel described in this section was done on two testmachines with the following hardware: Athlon XP (Palomino) 1.4 Ghz, with 256 Mb memory and connected by a 100 Mbit network. The machines were running the Debian Linux operating system, kernel version 2.6.11.3. We used the JBoss 4.0.5.GA Application Server and Java Virtuele Machine 1.5.0 10-b03 to run the different Java based test components. The latency and throughput tests were done using a 1 minute warm-up period, followed by 5 minutes of measurement.

Figure 7 shows the results for this experiment. The first thing that we can notice on this figure is that the event channel is very scalable and latencies increase as message sizes go up. The latency remains almost constant at 4 msec until a sendrate of 18000 events/sec for all events payload sizes. When this injection rate is reached latencies increase very rapidly until the maximum sendrate is reached.

A second experiment focuses on the throughput measurement of the external communication. This metric is defined as the maximum number of messages that can be delivered in a time period. Figure 8 shows the used test setup. The sending component sends messages to itself using the messaging system, which is located on a different machine. The messages transmitted in a time period increases until there is a difference between send rate and receive rate.

Table 1 presents the results for this experiment. Because the events are transmitted through a 100 Mbit network we did a comparison between the theoretical and practical achievable sendrates. The theoretical sendrate is calculated using the formula: $SR_T = NC/ES$. Where $SR_T$ presents the theoretical sendrate, $NC$ the network capacity in KiB/sec and $ES$ the event size in KiB.

As can be derived from the comparison table, there is a significant difference when sending events at a high rate. When we measured the time the senders blocks while sending an event, we notice an increased timeout with higher event payload sizes. Therefore, we can conclude the current approach of using non-blocking sockets resulted in behavior which is not truly asynchronous. When sending events between containers residing on different machines, the network becomes the bottleneck.
Table 1: Theoretical and practical sendrate comparison when sending events between different containers located at different testmachines.

<table>
<thead>
<tr>
<th>Event Size (Kb)</th>
<th>Theoretical sendrate (events/sec.)</th>
<th>Max connection call delay (ms)</th>
<th>Max sendrate (events/sec.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,826</td>
<td>15496</td>
<td></td>
<td>0,46</td>
</tr>
<tr>
<td>5,826</td>
<td>2197</td>
<td>0,90</td>
<td>1111</td>
</tr>
<tr>
<td>50,826</td>
<td>252</td>
<td>6,34</td>
<td>158</td>
</tr>
<tr>
<td>500,826</td>
<td>26</td>
<td>173,01</td>
<td>6</td>
</tr>
<tr>
<td>2000,826</td>
<td>6</td>
<td>284,42</td>
<td>4</td>
</tr>
</tbody>
</table>

7. Future Work

The evaluation results presented in this paper show the event architecture to be capable of high throughput event processing. Due to the behavior Java Virtual Machine memory management and the use of automatic garbage collection we assume the obtained results can be further improved upon using Garbage Collection tuning [19].

Hosting business applications are typically outsourced to specialized hosting providers. This reduces the maintenance costs and investments for the application provider. To accommodate the hosting provider, the platform should support autonomic management and load balancing to make optimal use of the available resources. We intend to incorporate the event bus in an event based distributed platform for Massively Online Virtual Environments [20], [21].

8. Conclusions

Today, the Java Enterprise Edition is commonly used as a component architecture where asynchronous communication can be achieved by the Java Message Service in combination with Message-Driven Beans. However, as event model it is not efficient enough. Alternative technologies exist but they all have similar disadvantages.

In this paper we presented the architecture for an efficient, generic event model. We also provided a working implementation using Java EE specific technologies like the Java Connector Architecture. JCA makes it possible to extend application servers with pluggable modules, which can provide connectivity to legacy information system or can extend the functionality of the application server (e.g. additional protocol stack). Because the Java Connector Architecture also specifies the communication between components residing in the same container we have chosen to use nonblocking sockets to realize the external communication.

From our evaluation results, we conclude that the predefined requirements are met. First of all, events can be easily exchanged in a asynchronous manner using publish/subscribe. The concept of logical buses, makes the model very flexible as components can register and unregister themselves at any time even if the specific event type was not produced yet. Additionally the event delivery is very performant. Finally, because the application server manages a pool of instances we can achieve a parallelized execution environment with increased performance.

Acknowledgement

Filip De Turck is a postdoctoral Fellow of the Fund for Scientific Research - Flanders (F.W.O.-Vlaanderen).

References