Experience with Lightweight Distributed Component Technologies in Business Intelligence Systems[⋆]

Leticia Duboc[1], Tony Wicks[1], and Wolfgang Emmerich[2]

[1] Searchspace Ltd.
80-110 New Oxford Street,
London, WC1A 1HB, U.K.
{l.duboc|t.wicks}@searchspace.com
[2] Dept. of Computer Science
University College London,
WC1E 6BT, U.K.
w.emmerich@cs.ucl.ac.uk

Abstract. Business Intelligence (BI) systems address the demands of large scale enterprises for operational analytics, management information and decision support tasks. Building such applications presents many challenges. They must support complex and changing data models, have fast turnarounds, present an up-to-date and accurate view of information and provide extensibility mechanisms for new analyses.
Widely adopted distributed object systems, such as J2EE can be heavyweight and inflexible when applied to the described scenario. This paper presents our experience when developing a data analysis system that applies a combination of lightweight distributed component technologies available for Java.
These technologies are combined in an event-based architecture that anticipates constant changes to analysis algorithms in short time frames and provides the ability to maintain correlated analyses in a consistent state. The resulting architecture is extensible, easy to deploy, highly configurable and has a very flexible data model. We compare this approach with existing distributed object systems and evaluate its suitability to provide business intelligence.

1 Introduction

BI encompasses a wide variety of tools and applications that can extract better business understanding from raw, typically transactional, data. This variety incorporates query and reporting tools, OLAP servers, data mining and data integration tools [3]. While traditional BI solutions are appropriate for many tasks, they are best aimed at the dimensions of a problem that remain relatively static. Operational analytics tools, instead, seek to better extract meaningful

⋆ This work is partially supported by tti Ltd. through KTP 3528.

information based on self-tuning and learning, resource conservation and dynamic expansion to the true dimensionality of the problem [2]. They should support complex data models, be extensible accumulate new analysis and allow for scalability.

We are interested in systems that will be capable of handling volumes in excess of 100 million transactions per day, accumulated over months or years. Scalability is often achieved by distributing computational tasks across a number of processors executing in parallel. This number can be increased to accommodate growing volumes, if the distributed software architecture has been chosen carefully. In enterprise settings, such software architectures are often implemented using distributed component technologies. Nevertheless, widely adopted distributed object systems, such as architectures based on the Java 2 Enterprise Edition (J2EE), can be inappropriate when applied to data intensive analysis scenarios [17].

The main contribution of this paper is an account of our experience when architecting PLUS, which is such an experimental environment to devise algorithms to be deployed in Searchspace's operational analytics solution. We initially investigated J2EE technologies, most notably the Enterprise Java Bean component model (EJB) [17] and the Java Messaging Service (JMS) [18]. We present reasons why these technologies do not do justice to the data-intensive problem domain and instead present a solution that uses more lightweight technologies. The main technologies used in this work were:

- Hibernate, an open source object/relational mapping toolkit for storing plain old Java objects (POJOs) to a database [20].
- Java Management Extensions (JMX), which provides management capabilities for a service-driven network [12].
- XDoclet, a meta-data template engine that parses the source code and generate artifacts such as configuration files and support code [21].

This paper is organized as following: Section 2 discuss some of the common used tools and techniques for data analysis. Section 3 presents PLUS, the real world system described in this paper, and its requirements. The following section discusses the inadequacy of J2EE for the problem described. Section 5 briefly introduces the software architecture that we have built using lightweight distributed component technologies. General observations and lessons learned are drawn in section 6. We then conclude the paper in Section 7.

2 Background

Extracting meaningful information from large data sets is challenging. Selection of suitable analysis approaches is non-trivial and comprises iterative processes of experimentation and testing. For numerous reasons, a single analysis algorithm will usually be split into a sequence of dependent steps. These steps reduce algorithmic complexity, allow intermediate results to be available for other purposes, such as user interrogation, and enable new analysis streams to use derived data that may already be available in the system. Additionally, division of algorithms has benefits in terms of system scalability and performance.

Layering analytics in this way introduces dependencies between algorithms and additional complexity in terms of managing dependencies associated with the data being processed. The challenge is therefore to create mechanisms that can manage these dependencies such that a system provides guaranteed, consistent results arising from changes to transactional, reference or other system data feeds. These features are necessary to force results to be re-calculated whenever analytics are changed and, more importantly, must be correctly managed to allow deployment into operational data changing environments.

A common approach is to use purpose built analytic tools, such as IDL [16] and MATLAB [22], which enable users to perform ad-hoc analysis. These tools, however, do not provide means to deploy created analysis in an operational environment. To reproduce a result with such tools, the analyst is forced to repeat the whole process. Further they are not designed to scale to the type of environment we are interested in.

An alternative approach is to create analyses that can be deployed operationally. Commonly, this would include database stored procedures and OLAP [23] tools, which are efficient and powerful query mechanisms. Nevertheless, those solutions, by themselves, do not provide means for distribution, sampling and parallelism. In addition, they may introduce portability problems.

Vendor specific approaches have other degrees of limitation. For instance, BusinessObjects is an analytical tool for summarization, visualization and reporting, not designed to be used as a framework to generate analysis that can be deployed operationally [5].

3 Problem Statement

PLUS is a data analysis environment that is used to extract meaningful information from large amounts of transactional data. The system comprises data loading, transformation and analysis. Report generation and information navigation (e.g. drill-down capabilities) is performed by an external system, the integration of which is beyond the scope of this paper. PLUS provides an experimentation environment for data analysts to create, test, execute and store analysis algorithms. Once defined and tested in PLUS, analyses can be deployed operationally on a business environment.

PLUS is currently used in the banking/finance domain in applications such as money laudering and fraud prevention. To date PLUS has been applied to analyze two million transactions over a historical period of two years, comprising over twenty gigabytes of data.

From an analyst point of view, PLUS is an experimental framework for the development of analysis algorithms. Based on data held within the system, or externally in a file, the analyst defines algorithms that are deployed into the PLUS framework. Analyses can be divided in logical stages, having intermediate results persisted for user consultation. Algorithms are stored by PLUS, so they can be re-executed whenever required. As an example, the analyst can produce from transactional data summary information that may be reused by later analyses. If new transactional data is added or a stage's algorithm is modified, the system automatically updates itself, maintaining a consistent state. Figure 1 illustrates those dependencies. Note that analyses can also be dependent on multiple previous results.

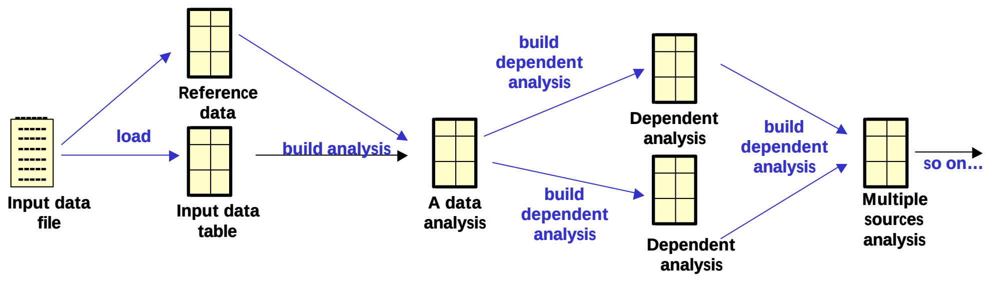

Fig. 1. Dependent Analysis

From an operational point of view, PLUS deals with large amounts of transactional data. Sampling capabilities allow analysis strategies to be tested before being applied to the whole set of data. PLUS also provides means to split analyses into logical and operational units of work. Analyses can be executed in parallel based on pre-defined analytical criterias. Dependencies are self-managed using inversion of control. Each analysis know its own dependency and this information is used by PLUS to compose individual components, so that changes can propagate through the system. Further, PLUS provides instrumentation through a web interface. Analyses parameters can be modified and their execution controlled.

We now introduce the main requirements for PLUS that led to the selection of the lightweight technologies described in this paper.

Support complex and changing data model: BI systems often maintain months or even years of detailed transactions. Analyzing such large volumes of data to identify trends, patterns and exceptions is a very complex process. PLUS required a rich data model that allowed algorithms to take advantage of object-oriented features, simplifing the analysis process. In addition, the data model had to be independent of the underlying database schema, offering a level of abstraction and simplifying its implementation.

Support for experimentation: When dealing with large amounts of data, it may not be clear from the outset how best to extract meaningful information. Analysts often try many different approaches in order to derive a comprehensive set of algorithms capable of obtaining relevant information from the data.

PLUS architecture had to provide efficient means for experimentation. Such features included support for fast development, deployment and test of analyses.

The work to define analysis algorithms had to be reduced to a minimum, ideally being tool-supported. The analyst was enabled to concentrate on the business logic and leave other time-consuming, non-core requirements to be addressed automatically by the system.

Ability to incorporate new analyses: To experiment efficiently with data analysis strategies, analysts had to be able to extend the system to support new algorithms with minimum effort. It was also a requirement that newly created analyses could be easily deployed and immediately incorporated, without affecting the running of the overall system.

Repeatable process and fast turnaround: In general, BI systems deal with very large amounts of data. Testing analysis algorithms against the whole data is, most of the time, inefficient and unnecessary. The system had to offer sampling capabilities and means to store algorithms that, once tested and tuned, could be applied to the whole data.

A second and more important reason for supporting storage and subsequent execution of algorithms was the dependent nature of analyses. This observation is further explained by the next requirement.

Maintaining consistent and accurate view of data: Load of new data and changes in algorithms will, almost certainly, impact downstream analyses. Immediately reflecting those changes is crucial, as BI systems should maintain a consistent and accurate view of information. Given the complexity dependencies may assume, manual update would be time consuming and error-prone.

PLUS had as a requirement to offer means to automatically update dependent analyses in face of changes. It is important, however, to have in mind that PLUS provides an environment for experimentation. Tests cannot impact already deployed analyses. PLUS had therefore to allow the deactivation of dependent analyses, while others are under test.

Execution control: Control over execution should offer, at a minimum, the ability to start, stop, restart, resume and change analysis properties.

Scalability and Performance: Support for scalability is not unique to BI systems. However, it is of significant importance, considering the large amount of calculations performed by algorithms. It was therefore a requirement that PLUS could accommodate new analyses and update existing ones with minimum impact on the system performance.

The requirements above demand a flexible, loosely coupled architecture that can be easily extended. A component-based architecture comes as a natural choice, allowing data analyses to be encapsulated by loosely coupled components that can be plugged into the system with relatively little effort.

4 Business Intelligence Systems with J2EE?

Initial investigations of the implementation considered the J2EE specification and, in particular EJB and JMS. At the time, like other Java developers, we went through a learning curve. Currently, many of the issues we faced are well documented in literature [19]. Our experience is summarized here:

– We considered using Entity Beans with container managed persistence for the persistence layer in order to achieve data independence from the database schema. This approach did not work because Entity Beans were designed to be stateful distributed components, not lightweight domain objects. In addition, despite the support for relationships, EJBs impose restrictions on the object format. Need for rich data models is not unique to BI systems and for all such cases EJB container-managed persistence represents according to [19] is limited by "tight coupling, obscure development models, integrated concerns and sheer weight". PLUS had as a requirement the ability to persist any Java object, so it could take advantage of OO/Java features, like inheritance and polymorphism. The use of EJB container-managed persistence was therefore discarded.
– The structure of database tables often does not match the structure of the logical entities they represent. Sometimes, even when objects differ from their exterior design, internally they still need to take advantage of the database structure to increase performance and achieve scalability. Examples include logically separated entities that map to the same denormalized table, a single entity that maps to multiple tables and given table (e.g. address) that is referenced by other tables (e.g. order and customer) with no explicitly relationship. CMP Entity Beans cannot naturally handle such cases [17].
– EJB QL allows additional finder methods to be defined in the Entity Bean home interface, associating each one of them with an EJB QL query in the deployment descriptor. This constraint means that the query logic had to be defined in the entity bean. For PLUS, which have different analyses using the same domain objects, the logic should be in the analysis class instead of in the entity bean. This approach does not only simplify the entity bean, but also makes analysis objects self-contained, providing a better understanding of the encapsulated algorithm.
– The EJB specification does not include management capabilities, despite the fact that it is implemented by some vendors. Using vendor specific features is not desirable, as it compromises the portability of the solution.
– JMS supports the asynchronous communication between distributed components. It is integrated with the EJB specification through message beans that are invoked when messages arrive on a queue. We considered implementing a publish/subscribe mechanism to maintain system concistency, using JMS and Message Beans. We concluded that the result would have been too heavyweight. JMS is intended for asynchronous communication between distributed components. Therefore, messages usually carry heavy payloads and have a significant deployment overhead for queues and their persistent

storage. We did not need these heavyweight mechanisms as all we required was to trigger re-execution of dependent analyses.

One can argue that EJB could be integrated with JMX for management purposes and with a flexible object persistence mechanism, such as Hibernate or Oracle TopLink, for the data modelling [15, 20]. For PLUS, however, JMX and Hibernate themselves are sufficient to address the system requirements, as explained in Section 5.2.

5 Implementing BI with Lightweight Technologies

5.1 Overview of Technologies

Selecting the appropriate set of technologies to implement an architecture that then fulfils the requirements is a challenging task, particularly since there is a complex inter-dependency between the use of particular forms of infrastructure and the architectures that they induce [8]. This section describes the technologies used in PLUS, along with the reason they have been chosen.

JMX: Java Management Extensions (JMX) provide the flexibility, interoperability, and dynamic management capabilities that are required for a service-driven network [12]. This work uses the JBoss implementation of the JMX specification. JBoss [10] is, itself, built around JMX. Our BI system architecture takes advantage of JMX, by implementing data analysis algorithms as JMX services.

JMX is particularly useful in the BI setting because:

- systems need to maintain an up-to-date and accurate view of information. Services can benefit from the JMX event-mechanism to re-calculate analyses if other services modify data they are dependent upon.
- analysis services can be hot-deployed into the JMX Server, being instantaneously recognized and incorporated into the system.
- the JMX instrumentation mechanisms allow a fine-grained control over execution and configuration of services.

Hibernate: Hibernate is an open-source object/relational mapping toolkit with facilities for data retrieval and update, transaction management and database connection pooling [20]. Hibernate was chosen in the PLUS architecture for the following reasons:

- BI systems usually have a complex and evolving data model. Unlike EJB, Hibernate provides a very flexible O/R mapping, designed to naturally persist objects following the common OO/Java idiom.
- PLUS strives for flexibility, giving the analyst the option to run analyses as simple standalone applications. Unlike the EJB persistence mechanisms, Hibernate can run from outside an application server, as it does not impose as many requirements on the objects to be persisted.

– Hibernate can be managed via a JMX Standard MBean, providing a convenient means to modify database related properties through the JMX console.
– Hibernate provides tools for code, mapping files and database schema generation. Shifting effort from labour intensive tasks, not only lets the analyst focus on business related problems, but also gives support for experimentation.

XDoclet: XDoclet, officially termed a "Javadoc metadata templating engine", parses metadata in Java source files and generates artifacts such as XML descriptors and/or source code [21]. XDoclet is a natural choice when using Hibernate, as mapping files and database creation scripts can be automatically generated from tags in the Java object to be persisted.

Sun has announced metadata/annotations in J2SE 1.5. This new feature provides the ability to associate additional data alongside Java classes, interfaces, methods, and fields. This additional data, or annotation, can be discovered at runtime using the Java reflection API [6]. For PLUS, the ability to query metadata at runtime would mean that some properties files would not have to be generated. J2SE 1.5 metadata, however, does not replace XDoclet as a code generator.

5.2 PLUS Architecture

Analyses dependent nature and support for experimentation and demand a flexible and loosely coupled architecture. A natural choice is to implement analysis stages as independent components that can be easily assembled. Communication is achieved through events, in a similar approach to SEDA [14], which consists of a network of event-driven stages connected by explicit queues. It combines aspects of threads and event-based programming models to manage concurrency, I/O, scheduling, and resource management needs of Internet services. The main distinction between both approaches is that while SEDA intends to support a massive amount of concurrent user connections, PLUS focuses on data processing.

Easy of deployment, extensibility and management of components is achieved by layering PLUS on top of JBoss implementation of JMX. Components are represented by JMX services that can be dynamically deployed, being immediately incorporated to the overall system. PLUS services are generic components deployed with a set of analysis algorithms. As JMX services, components are exposed by JBoss JMX console for instrumentation.

Hibernate is placed between the JDBC layer and JMX services. It provides a level of abstraction, allowing POJOs to be persisted in the database. Free from constraints in input and output objects, analyses can take advantage of a rich data model and OO features like inheritance and polymorphism. The use of Hibernate is, however, not enforced. Analyses can have access to the underlying JDBC layer if desired. Fig. 2 gives an overview of PLUS architecture.

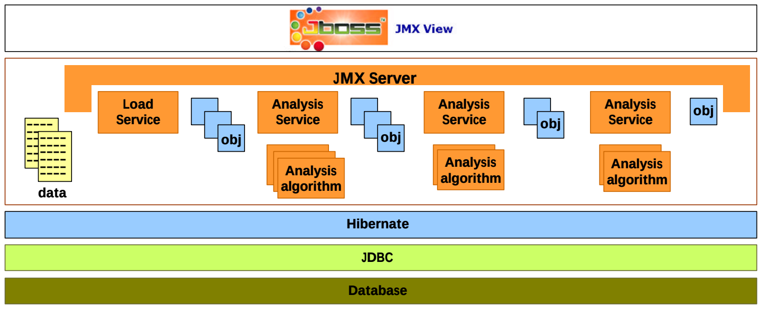

Fig. 2. System overview

Services and Notification: In comparison with other distributed technologies which often lead to complex interfaces between components, message or event orientation creates a small number of simplified programming interfaces [9] [11]. This interfaces can be widely applied as the are not dependent on underlying functionality and comprise simple message or event handlers, which allow simple system re-configuration [7].

Given the amounts of data handled by PLUS, passing analysis results in messages would be prohibitively expensive. In addition, analysis results are by themselves useful information, which should be persisted. Hence, there is no need for direct interaction between components. Analyses algorithms have knowledge about their input and output data format, but are completely unaware of how the input data has been produced. Simple events that informs of changes in the input data are more appropriate than complex messages.

PLUS implements events through the JMX publish/subscribe mechanism. Once notified, the service can update its analysis accordingly and inform other services that are dependent on its results. Updated analysis results are therefore communicated through the system in an asynchronous way, maintaining correlated analyses in a consistent state.

To receive notifications, a service needs to register as a listener of other services that affect data it is dependent on. This includes already deployed services and others that may be deployed in the future. The assembly of components is implemented through inversion of control. Services have knowledge of the tables/hibernate objects they are dependent on. PLUS uses this information to assemble components as publishers and subscribers. In addition, a newly deployed service registers with the MBeanServer to be informed about the future deployment of services that can change its input data. This mechanism guarantees that dependencies are self-updated whenever a new service is deployed into the system.

Furthermore, PLUS architecture provides a mechanism to avoid unnecessary work when a notification is received. A service can recognise the changes in the input data and update only the affected database rows or persistent objects in the result. The ability to automatically maintain a consistent state offers a significant support for experimentation. Hence, users do not have to worry about downstream analyses that might be affected by the newly incorporated algorithm.

Definition of analysis algorithms: As an experimental environment, PLUS should provide a fast turnaround. Users must be able to quickly develop new analysis strategies and understand existing ones. Code generation plays an important role in rapid development and transparency. It can be used to avoid code duplication and transparent models [19].

PLUS adopts a code-centric approach for the implementation of data loading and analysis algorithms. This method requires from the analyst the implementation of a single Java task class. Each task encapsulates one analysis algorithm. Tasks read data from an input source (e.g. file or tables), perform the required computations and persist the results. The analyst embeds related meta-data in the javadoc comments of the class source code using XDoclet tags. Meta-data represent relevant information about the task, such as analysis specific properties, output data format and dependencies. Automatic generation of code and support file encourages experimentation by considerably reducing the amount of work done by the analyst.

To illustrate, Fig. 3 shows the XDoclet meta-data in a task's code. In this example, the task is dependent on the data in the `BranchVolFin` table and generates its result in the `DistinctBranchVolFin` table. The output data format is explicitly defined in the `task.output.java` tag. The `control-entities` tag is an example of an analysis specific property.

```
 /**
 * @task.input input-entity="BranchVolFin"
 *            control-entities="AnalysisFin.productId"
 *
 * @task.output.java name="DistinctBranchVolFin"
 *                  fields="batch:long:6,
 *                  productId:long:10,
 *                  transactionType:long:7,
 *                  analysisType:String:20,
 *                  score:double:10"
 */
public class DistinctBranchVolFinTask extends ServiceTask {
  ...
}
```

Fig. 3. Meta-data in task code

XDoclet is used to automatically generate properties and configuration files, Hibernate persistent classes, mapping files and even database tables. Most of the code/file generation process did not have to be developed, as it came for free with Hibernate and its integration with JMX. Fig. 4 illustrates part of the output Hibernate object that XDoclet derived from the tags in Fig. 3.The Hibernate object is simply a javabean object with XDoclet tags defining the format of the columns in the database. The Hibernate mapping file is an XML document that maps fields in the Java class to columns in the database. Tables themselves are automatically created when analyses are deployed into the PLUS framework.

One can argue that code generation, as used in PLUS, combines concerns that should be logically separated, such as code and database schema design. Coupling code and configuration is certainly a downside. We have however opted for this approach because, as an experimental environment, PLUS should provide a fast turnaround. Having output object definitions as XDoclets tags in the task code, helps not only in the generation of Hibernate code and database schema, but also gives a better understanding of the analysis logic itself.

```
public class DistinctBranchVolFin {

  /** Independent Identifier **/
  private long id;

 /** Batch id for this load **/
  private long batch;

  /** Output Fields **/
  private long productId;
  private double score;

  /**
   * @hibernate.id column="ID"
   *              type="long"
   *              length="10"
   *              generator-class="native"
   */
  public long getId() {
    return id;
   } ...
```

```
<?xml version="1.0"?>
<!DOCTYPE hibernate-mapping PUBLIC
 "-//Hibernate/Hibernate Mapping DTD
2.0//EN" "hibernate-mapping-2.0.dtd">

<hibernate-mapping  schema="tdb">
  <class name="DistinctBranchVolFinTask"
         table="DistinctBranchVolFin"
         schema="tdb"
         dynamic-update="false"
         dynamic-insert="false">

      <id name="id"
          column="ID"
          type="long"
          length="10">
          <generator class="native"/>
      </id>
...
</class>
</hibernate-mapping>
```

Fig. 4. Generated Hibernate Object

PLUS does not enforce any restriction in the way data is retrieved, processed and stored by analysis tasks. Analysts may choose to use, for example, the Hibernate query language or direct JDBC. It is worth mentioning that, given the complexity of data in BI systems, Hibernate will provide many advantages. The very powerful object/relational mapping offered by Hibernate allows the task to take advantage of object-oriented features, like inheritance, association, composition and collections. The use of Hibernate also offers an abstraction layer, simplifying the task code, and other facilities, like transaction management, database connection pooling, programmatic as well as declarative queries and declarative entity relationship management. As an example, consider distinct Hibernate objects representing credit and debit card transactions. Having both extending from a common card transaction object, an analysis can take advantage of polymorphism to easily derive all card transactions in a given retailer chain during the summer sales.

Experiments have shown that there is a small performance penalty for using Hibernate. Nevertheless, specially in cases where the performance is constrained by the analytical work, this penalty is not the overriding factor when opting for one or the other approach. In an experiment, an analysis processing 10,000 rows took 123.47 seconds with JDBC and 140.19 seconds with Hibernate.

Execution and Instrumentation of Analysis: Scheduling and ordering of events is an important concern when using a staged event-driven approach. In approaches like SEDA [14], stages are responsible for defining their own scheduling policy for incoming events. Examples of policies are FIFO (First In, First Out) and SRTP (Shortest Remaining Process Time). PLUS uses events to maintain a consistent state between correlated analyses. The framework is not designed to support a large number of concurrent user connections, not having fairness in response time as a major concern. For this reason, PLUS uses a simple FIFO policy, having events processed in the order they arrive.

The event-based model is combined with thread level concurrency to enhance performance. Despite encapsulating a single algorithm, many task instances can be run simultaneously for different subsets of the data. The criteria for splitting the work to be done into tasks is data analysis specific, since the algorithm has to be consistent with the data set. For example, in a summary algorithm that requires the month of a given transaction, it is reasonable to partition the calculations according to the months, but not to days.

To handle the large amount of calculations required, PLUS uses a worker / task / controller architecture [13]. Workers are thread objects that execute tasks. The number of workers, as well as other properties, can be dynamically set and controlled through the JBoss JMX console. The analysis work is split into several tasks, which are stored in a task provider and individually supplied to workers. The controller creates and manages the execution of workers. It is also the controller's responsibility to respond to instrumentation requests (start, pause, resume, complete, fail) sent through the JMX console. Fig. 5 illustrates this architecture.

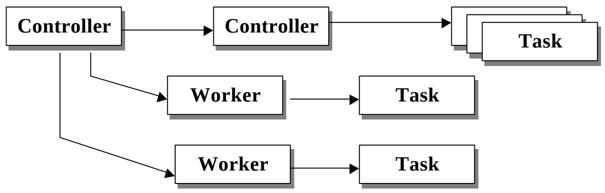

Fig. 5. Worker/Task/Controller Architecture

Tasks are themselves Hibernate objects that are persisted in the database, as a "to do" list. Every time a task is completed, its Hibernate representation is removed from the persistent storage. Duplicate tasks are not persisted, so work is not unnecessarily executed. Adding this persistent nature to tasks only required an auto-generated mapping file and database table. This approach adds robustness to the solution, allowing the system to restart from where it has been paused and recover in case of a failure.

Instrumentation of components is yet another feature provided for free with JMX. Public methods of a service are automatically exposed through the JMX console, offering the data analyst fine control over the execution. Users can monitor task's execution, set analysis specific properties, stop, start, restart, resume and change the level of parallelism of services. Futhermore, the Hibernate integration with JMX provides the ability to modify the JDBC datasource properties, include/exclude mapping files and other features.

Scalability and Performance: Event based systems present a range of possibilities for increased performance, data redistribution and operation. Implementations can move to more performant realizations that use clustering, partitioning or other methods to bring system capabilities benefits. Having the system composed by self-contained components allows the use of clustered compute resources that are available with most application servers. In our case, we can use the JBoss clustering mechanism and apply it to the MBeans that execute services in order to exploit real parallel execution (as opposed to interleaved concurrency on the same CPU).

6 Lessons Learned

PLUS architecture evolved over time. This work involved investigation of technologies, experimentation with different designs and close interaction with analysts. Our experience is summarized below:

1. Persuasive industry driven technologies are not always the best solution. Market leaders like Sun, BEA, Oracle and IBM are aggressively pushing the use of "golden hammer" [4] [19] technologies like EJB. For PLUS, and systems with similar characteristics, full J2EE solutions represent a high-overhead with little benefits. Our experience with PLUS has shown that it is possible to leverage benefits from lightweight open source projects and industry standard mechanisms for component interoperability and communication. Hibernate, in particular, has proven to be able to handle complex data models required by BI systems.
2. Having dependencies directly handled by the system removes the need for scheduling, improving maintainability and management. Our previous experience in coordinating dependent components through scheduling mechanisms have proved to be error-prone and hard to configure and maintain. Analysts were required to have an overall knowledge of the complete system, so they could manually update dependencies when adding or removing components. PLUS makes use of hot-deploy and inversion of control to assemble components into a publish/subscribe model. Through information provided by the components themselves, PLUS is able to automatically update dependencies whenever they are added or removed from the system. This approach requires less knowledge to deploy a module, as analysts only have to be aware of the algorithm being developed and its input data format.
3. JMX provides a flexible approach for internal, and potentially external, management. JMX instrumentation has added significant support for experimentation, as it allows changes to the system configuration at runtime. This feature also provides means for future optimizations, such as automatic resource allocation for stages. In addition, as JMX is an open standard, PLUS can

be directly integrated with external management systems, such as Tivoli [1], and operational environments.

4. A fundamental lesson learned was the importance of an optimal configuration. A staged event-based architecture potentially increases the performance of the system. However, the potential latency caused by both the granularity of the tasks and the sizes of the data chunks processed by stages could yield very poor performance. If stages were fully executed one after the other, the time to completion would comprise the sum of individual stages execution times. PLUS logically splits analyses into units of work, firing events when individual units have been completed. This approach enables dependent components to start execution before previous stages have been completed, creating a pipeline effect. Maximizing the throughput of this pipeline requires a careful design of analysis algorithms and eventing strategies.
   Consider an example where the input data represents transactions on a given retailer chain. A transaction has, among others, the following fields: “batch identifier”, “month of the purchase” and “product identifier”. For this example, assume that the two first fields coincide. Events informing changes in the transactional data are based on the “batch identifier”. An over-simplified analysis defines the number of products purchased by month. The experiment started with 884,321 transactions, 10 batches and 100 different products. An extra batch of 124,479 rows was added to the input table, forcing the analysis to be recalculated. The following settings were tested:
   - First, the analysis partitioned the work to be done in parallel tasks according to “month of purchase”. This is a sensible choice, since the batch coincides with the month of the transaction. The addition of a new batch generated a single task that processed only the added batch. In this case, the analysis completed in 10 seconds.
   - In a second run, the work was split according to the “product identifier”. This is not an appropriate selection, as the algorithm is based on months, not products. Having tasks selecting the input data by product identifier unnecessarily recalculates the number of purchase for every month. This setting generated 100 tasks, one for each product identifier, and took 126 seconds to run.
   - In a third experiment, the work was divided according to both “product identifier” and “month of purchase”. This can be a reasonable choice if the analysis component wants to take advantage of parallel execution of tasks. In this case, again 100 tasks were instantiated, but the work was only performed in the added batch. However, for this small sample of data, 100 tasks were still an unnecessarily large number. Setting the task to use a “modulo 10 function” on the product identifier creates 10 tasks which complete in 8 seconds.

   There is a need for careful design of both architecture and analysis algorithms to maximize process usage and gain scalability. As illustrated by the example, selection of tasks split criteria will be analysis dependent. Initial setting usually follow guidelines and final tuning can be done through experimentation.

5. Although we talk about PLUS as a single system, it is actually a framework composed of four independent sub-projects. The adoption of sub-projects has improved the development as their external use has helped to generalize requirements. Further, this approach has considerably improved PLUS configuration management.
6. As to any tool, PLUS has been carefully designed to provide information and encouragement for the data analyst. We believe that part of the system success was the tight interaction of developers and analysts operating as users.

7 Conclusion

This paper has described PLUS, a data analysis environment to extract meaningful information from large amounts of data. PLUS can be classified as a BI system and more broadly as a data-intensive application. In such systems, the data process may be split into a sequence of dependent steps. These steps reduce complexity and allow intermediate results to be available for other purposes.

PLUS uses an event-based architecture where large amounts of data are processed by dependent stages. Having stages defined by loosely coupled and self-contained components allows for modularity, extensibility and scalability. Stages can be added and redefined for optimal configuration. Performance is improved by in-process threading mechanisms within stages, having tasks parallel executed in different subsets of the data.

Dependencies in the proposed architecture are handled through a publish / subscribe event mechanism. Staged execution is triggered by events produced by previous phases, freeing the system from providing scheduling mechanisms. Furthermore, dependencies are self-managed, having stages automatically assembled through inversion of control.

Instrumentation also plays an important role in PLUS. Finding the optimal configuration may prove to be challenging. The ability to experiment, by controlling the execution of stages and tuning of data process specific properties, is a valuable feature for complex data analysis systems.

The proposed architecture caters for experimentation, allowing the system to accommodate changes either in short periods or over time. Finally, the realization of the design through the implementation of a real-world system has proven the adequacy of lightweight technologies to large scale data processing applications.

Acknowledgments

The authors would like to acknowledge Dave Martin, Nicola Harris and Alok Rana for their support in both development and review of this work.

References

1. Tivoli: Intelligent management software for the on demand world. Technical report, http://www-306.ibm.com/software/tivoli/.

2. Searchspace: Enabling the Intelligent Enterprise. Technical report, April 2003.
3. Software Scoops - Insights on Software. Technical report, August 2004.
4. William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., 1998.
5. Business Objects. Businessobjects query and analysis. Technical report.
6. Calvin Austin. J2SE 1.5 in a Nutshell. Technical report, http://java.sun.com/developer/technicalArticles/releases/j2se15, 2004.
7. G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th international conference on Software engineering*, pages 261–270. IEEE Computer Society, 1998.
8. E. di Nitto and D. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *"Proc. of the* 21^{st} *Int. Conf. on Software Engineering, Los Angeles, Cal."*, pages 13–22. ACM Press, 1999.
9. Lyman Do, Prabhu Ram, and Pamela Drew. The need for distributed asynchronous transactions. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*, pages 534–535. ACM Press, 1999.
10. Marc Fleury, Scott Stark, and The JBoss Group. *JBoss Administration and Development.* John Wiley and Sons, Inc., 2002.
11. Ann Wollrath Samuel C. Kendall Jim Waldo, Geoff Wyant. Events in an rpc based distributed system. In *USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems*, Mountain View, California, USA, 01 1995. USENIX, Sun Microsystems Laboratories.
12. Juha Lindfors, Marce Fleury, and The JBoss Group. *JMX: Managing J2EE with Java Management Extensions.* SAMS, 2002.
13. Anoop Mangat and Iain McLaren. Personal Communication, August 2000.
14. Matt Welsh and David E. Culler and Eric A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Symposium on Operating Systems Principles*, pages 230–243, 2001.
15. Kirk Pepperdine. Oracle9iAS/TopLink By Example. Technical report, http://otn.oracle.com/oramag/webcolumns/2003/techarticles.
16. Research System Inc. The interactive data language. Technical report, http://www.rsinc.com/idl/.
17. P G Sarang, Kyle Gabhart, Andre Tost, Tim McAllister, Rahim Adatia, Matjaz Juric, Ted Osborne, Faiz Arni, Jeremiah Lott, Vaidyanathan Nagarajan, Craig A. Berry, Dan O'Connor, John Griffin, Aaron Mulder, and Dave Young. *EJB Professional.* Wrox Press Inc, 2001.
18. Sun Microsystems. Java message service specification 1.1. Technical report, http://java.sun.com/products/jms/docs.html.
19. Bruce A. Tate and Justin Getland. *Better, Faster, Lighter Java.* O'Reilly Media Inc., 2004.
20. Hibernate Team. Hibernate reference documentation 2.1.4. Technical report, http://www.hibernate.org/hib_docs/reference/en/html/.
21. XDoclet Team. Xdoclet: Attribute oriented programming. Technical report, http://xdoclet.sourceforge.net/xdoclet/index.html.
22. The MathWorks. Matlab tutorial. Technical report, http://www.math.ufl.edu/help/matlab-tutorial.
23. The OLAP Council. Olap and olap server definitions'. Technical report, http://www.olapcouncil.org/research/glossary.htm.