Scheduling and Optimising XML Pipeline Processing

Michal Sankot

Submitted to Council for:

Master of Science (Research)

Institute of Technology, Sligo

Supervisors:

Tom McCormack

Tony Partridge

Submitted to the Higher Education and Training Awards Council, July 2004

Declaration

Declaration

This thesis has not previously been submitted to this, or any other college. With acknowledged exception, it is entirely my own work.

Michal Sankot

[illegible]

ABSTRACT

Abstract

Scheduling and Optimising XML Pipeline Processing, Michal Sankot

This thesis describes PropelXbi – an implementation of the XPipe paradigm – then investigates and critically assesses relevant techniques which have the potential for streamlining PropelXbi's performance and, finally, it presents and tests improvements in PropelXbi which are achieved by implementing a number of devised enhancements.

XPipe is a paradigm for processing a great number of very large XML documents in an efficient way. PropelXbi is a commercial implementation of the XPipe paradigm based on JMS and J2EE architecture. Relevant topics we have investigated include architectures and enhancement techniques used in parallel processing, Jackson Inversion, TupleSpaces, Project JXTA and Grid computing technologies. We have implemented a J2SE-based compact version of PropelXbi runtime (compiled pipelines) and a Grid-based distributed version of PropelXbi. Tests showed that the compact version of PropelXbi runtime achieves significantly better performance than original J2EE version. Tests also showed, that distributed processing can be used for streamlining PropelXbi's performance and that the distributed version follows the same laws as other standard parallel processing systems. This thesis identifies potential enhancements from different areas of computing which can be used not only for streamlining PropelXbi, but also for any other similar large-scale document processing system, and demonstrates that they can be efficiently utilised.

PREFACE

Preface

This thesis has three major goals:

1. To present the XPipe paradigm and its implementation called PropelXbi; to examine the current landscape of document processing and then to identify the position of XPipe paradigm in it.

2. To scrutinize techniques, used in different areas of the computer world, which are related to document processing and examine if these ideas can be used to enhance the performance of PropelXbi.

3. To examine what enhancements are actually delivered when selected techniques are implemented in the existing application.

The text of this document is organised correspondingly so that every part corresponds to one of the above goals.

PART 1 – PropelXbi and Document processing analysis

This part gives a description of the XPipe document processing paradigm and presents the PropelXbi implementation. Afterwards, it examines document processing in general and identifies the position of XPipe within it.

Chapter 1 explains the concept of XPipe processing and presents a detailed description of PropelXbi, which is its commercial implementation.

Chapter 2 examines the different document processing scenarios that are available taking into consideration the number of documents that are processed at the same time and the number of processors available. It categorises these scenarios; states the advantages and disadvantages of each and identifies which scenarios are relevant to the XPipe paradigm.

Chapter 3 surveys the document-processing techniques which are currently used and establishes the position of XPipe among them.

Chapter 4 examines a study which aimed to precisely characterise the computing related process of complex problem solving, and then looks at XPipe and PropelXbi in light of the results of this study.

PART 2 – Document processing techniques survey

This part investigates different techniques related to document processing, which are used in the computer world and appear to have the potential to offer ideas which can be used to enhance the performance of PropelXbi.

Chapter 5 looks on the relevant improvement techniques used in parallel processing. Firstly, it examines the architectures used for high performance computing, then it introduces parallel problem classes and, finally, it discusses techniques which can be utilized in pipeline document processing.

Chapter 6 examines the Jackson Inversion technique. At first, it presents Jackson Structured Programming in which the Jackson Inversion technique is found. It then examines how Jackson Inversion can be used in XPipe's implementation and, as a final point, it introduces the concept of the XComponent compiler – an implementation of Jackson Inversion in PropelXbi.

Chapter 7 scrutinizes the different technologies used in area of distributed computing which relate to our case of document processing. The concept of Tuple spaces, Project JXTA and Grid computing technologies are researched. For each technology, their main concepts are introduced, their implementation is described and finally their relation to PropelXbi is discussed.

Chapter 8 gives an overview of the techniques and concepts researched in chapters 5 to 7. First, the current state of PropelXbi is described and then each of the surveyed techniques is summarised in terms of the way they can enhance current pipeline XML processing.

PART 3 – Document processing enhancements implementations

Part 3 looks on implementations of the document processing enhancements found in Part 2 and examines what enhancements are really delivered when selected techniques are implemented in the existing application. The last chapter summarises the results of this thesis and suggests directions for future work.

In Chapter 9 we present the implementation of XComponent Compiler conceived in Chapter 6. We explain its concept; describe the technical design and implementation and present a performance comparison of PropelXbi and pipelines compiled by XComponent compiler.

Chapter 10 examines the implementation of the distributed document processing system. First, we present the implementation of distributed system, then we present the questions we wanted to answer about the utility of the document distributed processing, then we give theoretical solutions to these questions and, in the end, we compare and contrast them with the results obtained from the performance tests of implementation of the distributed compiled pipelines processing system.

Chapter 11 summarises the major findings of this thesis and suggests directions of potential future work.

TABLE OF CONTENTS

Table of Contents

TABLE OF FIGURES

Table of Figures

PART 1

PROPELXBI AND DOCUMENT PROCESSING ANALYSIS

CHAPTER 1

XPIPE AND PROPELXBI

1 XPipe and PropelXbi

In this chapter we describe the XPipe methodology of document processing and present a detailed description of its implementation, called PropelXbi. XPipe is described in section 1.1 and PropelXbi in section 1.2.

1.1 XPipe paradigm

XPipe is a methodology for document processing that was developed by Sean McGrath, CTO of Irish IT company Propylon (Redmond & McGrath 2002). Its main aim is to define a standard methodology for large-scale document transformation processing in an efficient way.

The XPipe paradigm is based on four essential concepts. These are listed in decreasing order of importance:

1. Break up the transformation into simple components (transformation steps/stages);

2. Chain the components into a pipeline, connecting them by queues;

3. Allow the components to be written in any language (they are black boxes, that only take a document in, process it and output it);

4. Describe the components using the XML language;

The first concept comes from observation that every complicated transformation can be decomposed into sequence of simple transformations. Interestingly, many of these simple transformations are used repeatedly and so, once these transformations are coded, they can be used again every time they are needed. Transformation decomposition also allows us to construct complicated transformations even in cases when it seems that writing one code for a whole transformation would be impossible. Components of the pipeline are, in the XPipe terminology, called XComponents.

The second concept naturally follows the transformation decomposition. Decomposition produces the sequence of components, which need to be chained in sequence again, in order to have components perform the intended transformation. In contrast with common code modular synthesis, pipeline chaining allows for more documents being present in the pipeline at the same time and thus more documents being processed at the same time.

As components may not be able to process documents at such speed, one document may not be processed before the next document arrives, queues are a natural storing space for incoming documents. With queues, the arrived documents simply wait in queue until the next component becomes free.

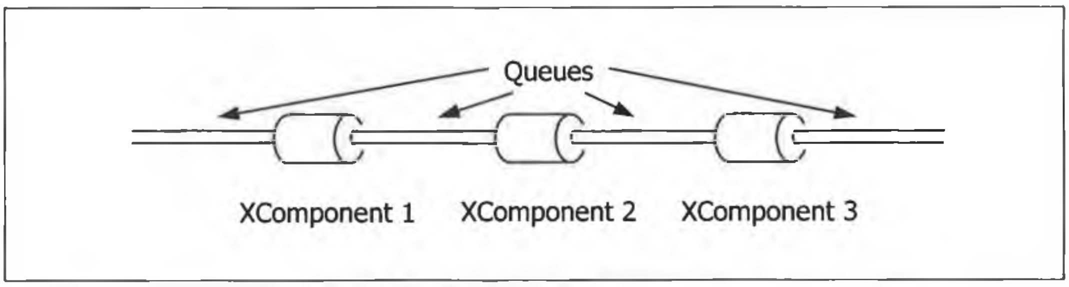


Fig. 1.1 Pipeline of XComponents

The third concept of not restricting components to any particular language gives XPipe great flexibility on how to process incoming documents. The view of the component is that of a black box, which takes a document in, processes it and flushes the transformed document out.

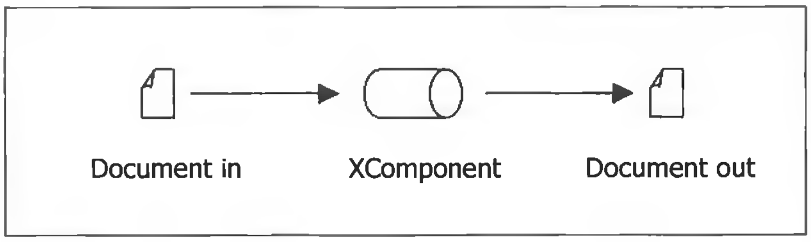


Fig. 1.2 XComponent model

Different languages and technologies (e.g. XSLT, Java-SAX, Java-DOM, Python etc) are good for different types of operations and this simple model of XComponent allows a component to be written in whichever language, best suits the transformation.

The last concept of describing XComponents with the XML language, aims at the possibility of processing the information in the XComponents by computer in a way that doesn't depend on the platform and the underlying data storage format. As the XML standard is commonly used today, there are many tools for processing this information in an easy and elegant way.

As said previously, the only limitation set on the XComponents is that they have to be able to take in a document, and output it afterwards. To create the complete document transforming device, the XComponents are connected together to create one big pipeline called XPipe.

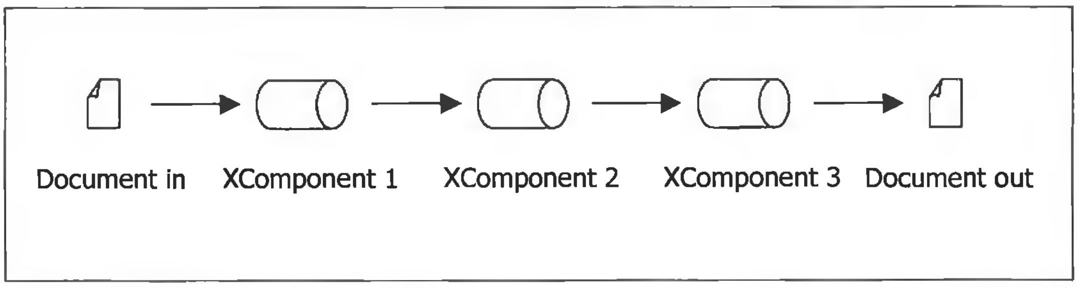


Fig. 1.3 XPipe model

XPipe also uses the concept of Scatter and Gather components. These components can be used when the document to process contains parts which can be processed separately. An example of such a document can be an overall list of all employees in a company, where each employee is represented by a large structure which can be processed separately. The Scatter component divides the document into a set of independent documents, which contain separated elements of the original file. After they are all processed, the Gather component assembles them together into the resultant document. By dividing the document into smaller independent pieces, the Scatter and Gather components allow the processing of documents, which would otherwise take a long time to process because of excessive size. Furthermore, by dividing the document, individual processing nodes can be better utilised, as division provides many small documents to process compared to a few original large documents which would use only these nodes to which they were sent and would leave all the other processing nodes unused.

As discussed in greater detail in chapter 3.3, the XPipe paradigm proves to have many advantages. The main benefits of the XPipe paradigm are:

- Transformation decomposition enables us to create very complicated transformations by a simple construction method;
- Components are language independent – each component can use the language and/or technique that best suits the transformation (XSLT, DOM, SAX, Python etc);
- Component structure facilitates easy load balancing;
- Decomposed structure allows for high parallelizability and scalability – mainly because of high independency of individual pipeline stages;
- Components are reusable – it uses the simple model of black box which takes document in, processes it and outputs it, and this allows simple reusability of previously written components;
- Enhanced monitoring capabilities – as the whole transformation is divided into clearly defined stages, it's possible to monitor the current stage of the document processing and whether there was a problem to find and in exactly which component it occurred;
- Easy maintenance – if one component needs to be changed, it can be simply plugged out, changed and plugged in again;
- Easy transformation changing – components can be easily added or removed as needed without the need to affect the whole code.

1.2 PropelXbi – an XPipe implementation

PropelXbi is Propylon's implementation of XPipe paradigm (Propylon 2003). It is written in Java, built on J2EE architecture (Sun Microsystems) with the use of

JMS messaging (Sun Microsystems) as a message passing communication system.

1.2.1 J2EE architecture

The J2EE architecture is based on objects called Enterprise JavaBeans (EJB's), which encapsulate some functionality, which can be used by calling the EJB's methods. This is different from other object-based architectures in that all the maintenance of EJB's is performed by the server and the programmer can focus on merely writing the logic of an object without concern about the maintenance tasks. Another benefit of J2EE is that the logic of EJB's is written as if there was only one object running even though in reality there are many objects running concurrently.

The server creates a pool of EJB's and, depending on requests from clients, assigns them to the tasks. If some of the EJB's aren't used for a long time, they are passivated and saved to the disk, so that more memory is available for objects demanded at the time. If they are requested again, they are re-activated.

All the object maintenance is hidden from the user who doesn't have to care about maintenance issues or about load balancing, which is done by the server automatically. As pooling is in the hands of the server, it can spread its functionality over more computers and all this remains transparent to the user.

There are three essential types of Enterprise JavaBeans, of which one particular – Message-Driven Bean (MDB) is used in PropelXbi. Message-Driven Bean is an object, which waits for the messages to come, and when they arrive, it carries out some action.

1.2.2 JMS architecture

An MDB receives messages from JMS queues, to which it can also send messages back. There are two types of queues. The first type is Multiple Publisher, Single Subscriber – simply called a "Queue", which delivers messages to one MDB. Messages can be sent to the Queue by any other EJB or normal Java program. This is equivalent to e-mails being sent to one person. The second type is Multiple Publisher, Multiple Subscriber, called "Publish/Subscribe", "Pub/Sub" or a "Topic". This is analogous to a News conference, where anybody can send a message to the queue, which is then received by everybody who subscribes to it.

1.2.3 PropelXbi architecture

The natural implementation of the XPipe paradigm would use a set of MDB's, implementing the XComponents, connected together with JMS queues. However, this would not allow for the dynamic assigning of MDB's to points in the pipeline where the worload is high and MDB's are most needed.

There is another problem set by the limitation of the J2EE architecture. It comes from the need to have the queues persistent, to have all messages saved in case the server crashes. Persistent queues can be created only at start-up of the J2EE server, and, at that time, it is not known how many queues would be needed, as the user can create and start a new pipeline at an arbitrary time after the start of the server. Maintaining a pool of persistent queues, which can be assigned as needed to the MDB's would be a solution. However, this is hindered by another limitation set by JMS architecture. An MDB can only be assigned to the queue at the time of creation of the MDB and it cannot be changed afterwards. This disallows use of a pool of queues, as there wouldn't be a way of assigning already created MDB's to them.

Because of these two limitations, the resulting PropelXbi architecture consists of one queue which caters for all messages being processed in PropelXbi. Messages which flow through PropelXbi contain an identification tag denoting the pipeline they belong to and thus it is possible to distinguish which pipeline they were submitted to, even though they are all held in a single queue.

The drawback of using one common queue for all messages is that monitoring is more complicated than it would be in case of separate queues for each component. Nevertheless, it is possible to implement a monitoring facility thanks to the information about what pipeline the message belongs. From a performance point of view, there is no difference in speed because queues are implemented as blocks in memory in both cases.

The processing part of the PropelXbi system is implemented by a pool of Message-Driven Beans, which all listen to the main queue. Documents are inserted into JMS messages which are then placed into this main queue. When an MDB detects a message, it retrieves it from the queue and performs the appropriate transformation of the document. Each message contains information stating which particular transformation should be applied on the document next and this data is used by the MDB to identify which XComponent should used to transform the document. Data contained in the message identifies which pipeline it belongs to, the last XComponent used to process it, the identification of the document it holds and the document itself.

Message-Driven Beans don't contain any transformation logic in themselves. When they retrieve information about what XComponent should be used, they ask the Executive (which is another Java object) which passes them the right component. This allows MDB's to be assigned to any XComponent that is needed at the time and thus it carries out load balancing by itself.
The whole PropelXbi architecture is depicted in the following figure:

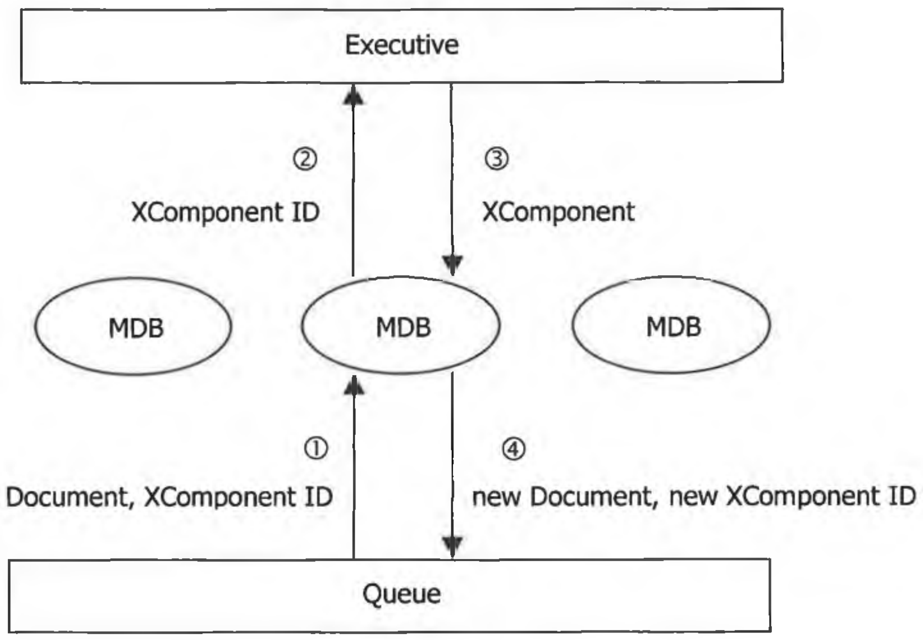


Fig. 1.4 PropelXbi architecture

Figure 1.4 shows the process of a document going through one pipeline stage – i.e. being processed in one XComponent. At first, the MDB retrieves the message from a queue, which contains the document to be processed and information about which XComponent to use. In the second step, it asks the Executive for the appropriate component and receives it in the following step. The MDB then executes the component it received and saves the transformed document back to the queue together with the information about the next XComponent, which should be applied to the document.

The presented architecture is a mid-level view of PropelXbi that is sufficient for a conceptual understanding how PropelXbi implements XPipe. The actual implementation is a little more complicated. The main queue is in fact implemented by four different queues. There is Input queue, which holds documents that arrived in the system and are to be put in the Processing queue. The processing queue stores documents waiting to be processed, and it is this queue that the MDB's watch for waiting documents. The third queue is the Error queue, where documents are placed which caused some error during the processing or which can't be further processed because of an error that occurred in some of the pipelines. The last queue is the Output queue holding processed documents ready to be shifted to their final destination.

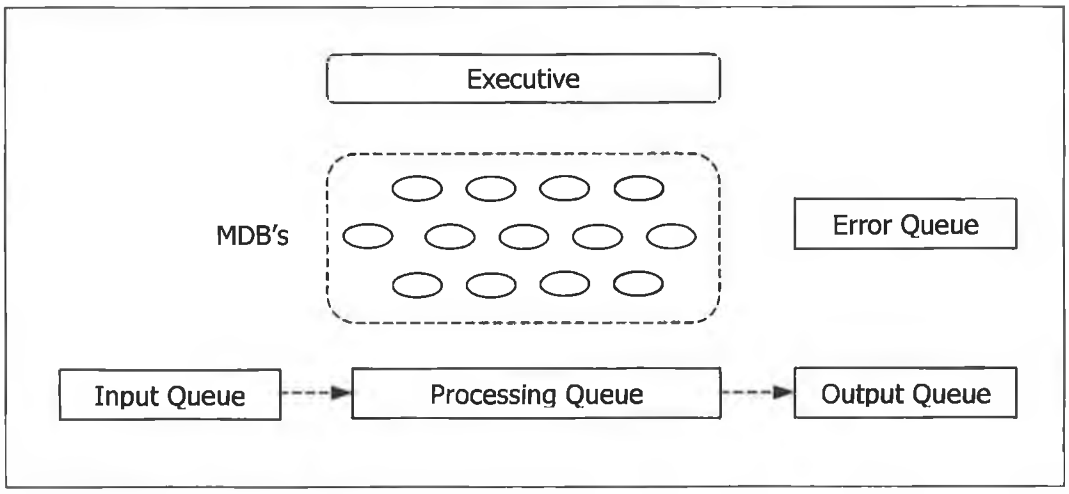


Fig. 1.5 Detailed view of PropelXbi architecture

Figure 1.5 shows the detailed architecture of PropelXbi. A document enters the Input queue, moves to the processing queue waiting to be processed. MDB's pick documents from the processing queue, process them and return them back to the same queue, when the document is fully processed it is output to the output queue.

All this implementation architecture is hidden from the user, to whom the whole PropelXbi system looks like the original XPipe design – like pipelines of components interconnected by queues.

The architecture of PropelXbi built on J2EE and MDB's has the benefit that the flow of documents and the load balancing are done automatically by the J2EE application server without any additional work required on the programmer's side. However, it is important to mention that the J2EE architecture doesn't provide any way to influence the way in which documents are passed between XComponents which in consequence means that we can't change the way scheduling is done in PropelXbi.

Chapter 2 analyses different scenarios of document processing and positions the XPipe paradigm within it. Chapter 3 then looks at different processing techniques currently used for document transformations and discusses the relation of XPipe to them. The last chapter of Part 1, Chapter 4 presents a project aimed at the characterisation of the complex problem-solving process related to computing and views the XPipe and PropelXbi in light of its findings.

Chapter 2

Document Processing Scenarios Analysis

2 Document Processing Scenarios Analysis

This chapter looks at different document processing scenarios, which exist in today's computer world.

At first, in section 2.1 we define the terms related to document processing which are used throughout the document. Section 2.2, then introduces a categorisation of the document processing scenarios and sections 2.3 to 2.6 look on individual scenarios in greater depth. Section 2.7 summarises the qualitative features of the inspected scenarios and juxtaposes them in graphical comparison and the closing section 2.8 lists the major concepts for improving efficiency which are employed in today's document processing.

2.1 Terminology definition

In order to describe different scenarios we first need to clearly set fixed terms, which will be further used throughout this documentation when speaking about document processing.

First, we define what will be considered a **Document**, which is the subject of the transformation processes examined in this thesis. A document is a group of data. The definition is general to allow us to speak about all types of documents without need of narrowing the spectrum of documents. In particular we set no conditions on how the data should be structured, ordered, whether it is local or remote, unique or replicated, or in any particular format. The document can be a HTML page, an XML page, a Word document, a raster or bitmap image, raw data gathered from a measuring device, etc.

Secondly, we define a **Processor**. It is a device which performs operations on a document. A processor can be understood as a general computing device or a computing node that performs operations on a document. Examples of processors can be an XSLT processor, code written by the user, an Enterprise JavaBean, ftp client, etc. In the following document, the terms 'processor', 'processing device' and 'processing node' will be used interchangeably.

We define a **Transformation** as one or more operations on a document. It is a higher-level task that we want to perform on the document. Examples can be converting a file from one format to another, annotating a document or copying a file from one folder to another. Each transformation consists of one or more lower-level operations. In the XPipe view of the world, a transformation is implemented by an XPipe pipeline.

To be able to examine transformations in greater depths we define **Operations**, which are logically independent elements of the whole transformation. An operation is a lower-level group of rudimentary actions that encapsulate one logically independent task that is to be done on the way to complete the whole transformation. When implementing the document processing system, the feature of logical independence of operations naturally leads to the concept of components that build up the whole transformation. In XPipe, an operation corresponds to an XComponent, which carries out one particular step of the transformation. An example of operations may be the adding of an element to an XML tree, removing comments from transformed code or saving a file to disk.

Each operation consists of three stages. These stages group actions of an operation and occur in a defined order. Every stage contains zero or more actions.

1. **Pre-processing stage**
   In this stage, the processing device prepares the document and the eventual resources for the main processing stage. It can be downloading a DTD file for validation of the XML document, connecting to a database or checking the input stream for a correct format

2. **Core-processing stage**
   The essential functionality of the operation is implemented in the core-processing stage. It can be, for example, applying an XSLT sheet, inserting a record from a database or converting the input stream to a format suitable for output.

3. **Post-processing stage**

   In the post-processing stage, the processor carries out actions that are necessary for successful and correct completion of the operation. For example, checking the output XML document for well-formedness, closing a database connection or adding standard formatting elements to the output steam.

The concept of three-stage processing with its distinct features does not have to apply only to stages of operations but can and will be used in context of distinct stages of transformations as well.

In the definition of operations we referred to **Actions**, which are rudimentary pieces of work carried out on a document. Examples of actions can be incrementing a counter, concatenating two strings or assigning a value to a variable.

The whole document processing model is demonstrated by following figure:

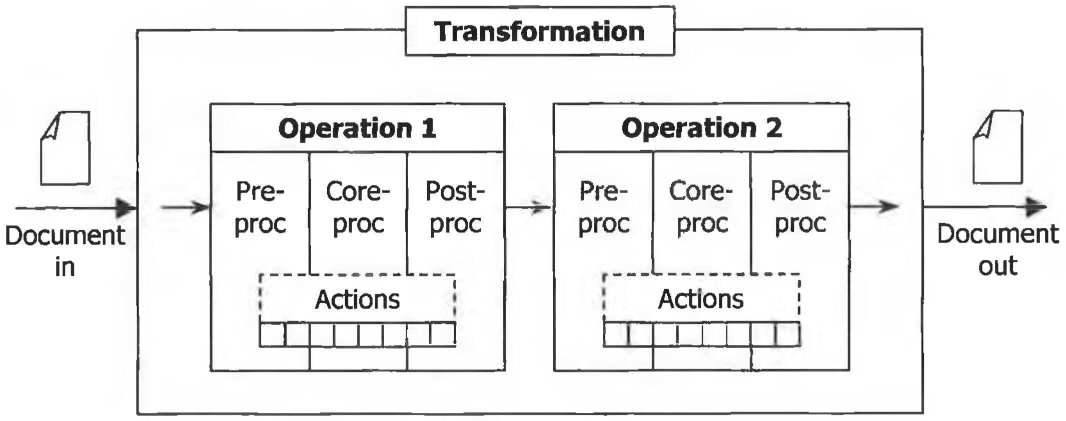


Fig. 2.1 Document processing model

Now, when we have defined all the needed terms, we can inspect how to categorise different document processing scenarios with regard to the unique characteristics of particular groups of scenarios and their specific features.

2.2 Categorisation of document processing scenarios

To divide the scenarios into different groups with different features we take into account the number of documents that can be processed at one time (D) and the number of processors available at one time (P).
This division leads into following four categories:

SDSP – Single document, single processor scenario
SDMP – Single document, multiple processors scenario
MDSP – Multiple documents, single processor scenario
MDMP – Multiple documents, multiple processors scenario

Examples could be:

SDSP:	applying an XSLT sheet to an XML file
SDMP:	distributed image processing
MDSP:	batch file conversion
MDMP:	pipeline document processing

A feature to note is that all SDSP and SDMP systems are inherently synchronous – when a new document is to be processed, it has to wait until processing of the previous document is finished and can enter only after it.

All four scenarios will be examined in detail in following sections. Section 2.3 describes SDSP, section 2.4 SDMP, section 2.5 MDSP and section 2.6 MDMP scenario.

2.3 SDSP – Single Document, Single Processor

Example: Applying XSLT sheet to XML file

The SDSP scenario is a case of document processing where only one document can be processed at a time and only one processor is available. SDSP document processing is also called a monolithic transformation, because the transformation process is viewed as one unit. This transformation process of SDSP scenario can be depicted by following figure:

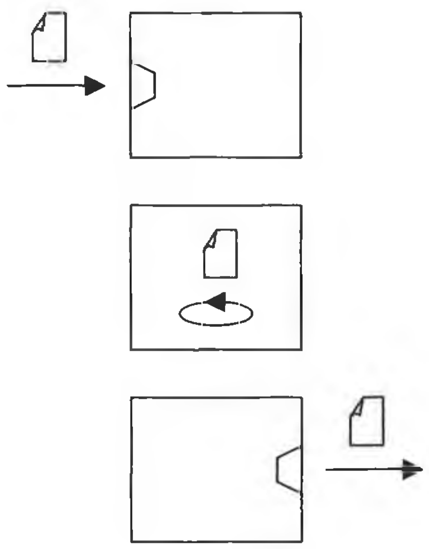


Fig. 2.2 SDSP scenario

As shown figure 2.2, one document enters the processing unit, is processed and then one document leaves.

The SDSP scenario by itself is not too interesting, but serves as a building block for more complex systems and also provides the basis for reasoning about scenarios which have SDSP's as its components. Even though it's not interesting from the point of view of structure complexity, it is a very common case in personal use, when the user needs to process one document just once.

Features of SDSP can be summarized by following list:

Advantages:	-	no or very little work needed to set up transformation
	-	no maintenance needed
Disadvantages:	-	inefficient for larger volumes

2.4 SDMP – Single Document, Multiple Processors

Example: Distributed processing of large image data

SDMP scenarios are common in situations where the input document is large in volume or the transformation is very computationally demanding or needs to be done very quickly. There are two approaches to SDMP processing employing pipeline and the scatter/gather concept.

2.4.1 SDMP – pipeline

Processing units executing smaller pieces of transformation are connected one after another. These units are SDSP nodes where output of the preceding component is passed to the input of the successive one. In contrast with the SDSP approach, the processing units in the pipeline paradigm do just a small piece of transformation and therefore their logic can be very simple. This allows for better re-use of components and easier monitoring.

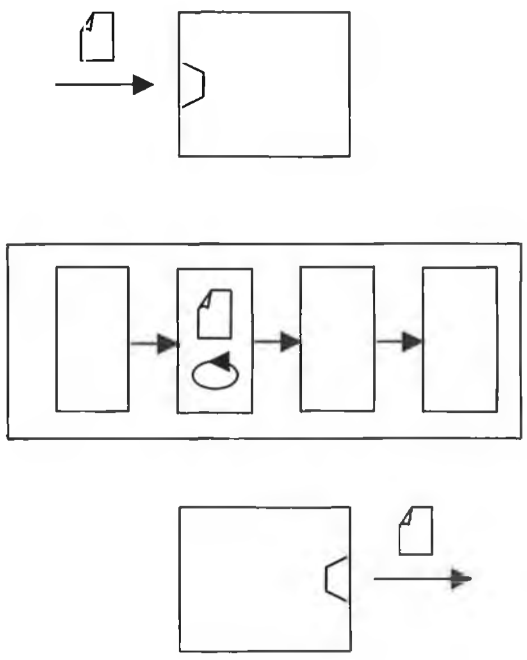


Fig. 2.3 SDMP – pipeline scenario

Apart from maintenance and monitoring advantages, pipeline processing doesn't bring improvement of execution speed. In fact processing in the SDMP-pipeline system can take slightly longer that in the SDSP system, as time spent by passing intermediate documents between components is added to execution time.

Advantages:	- fast assembly of complicated transformations
	- good monitoring capabilities
	- support of re-use of components for future transformations
Disadvantages:	- work needed to assembly pipeline
	- slightly longer time of completion than SDSP

2.4.2 SDMP – Scatter/Gather

In the Scatter/Gather approach, the document is divided (scattered) into small parts which can be processed in parallel and these document portions are distributed on the available processing nodes. Usually, not all the document can be processed in parallel and so, there is a part of the document that has to be processed sequentially without distribution to available nodes.

In the group of executing nodes, there are three essential units. The scatter unit examines the input document and distributes its parts to the available nodes. The core-processing unit is the group of nodes carrying out core-processing of the transformation. These nodes can be simple SDSP nodes, without any knowledge of being part of a Scatter/Gather system. Finally, the Gather unit, which brings together individual transformed document parts to assemble the final complete transformed document.

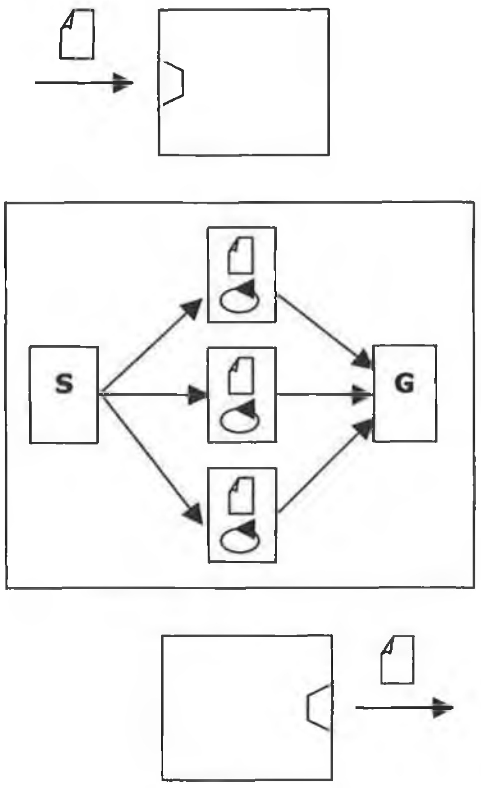


Fig. 2.4 SDMP – Scatter/Gather scenario

In Appendix A, we investigate the efficiency of using the Scatter / Gather approach in detail and reach the following conclusions:

The speed gain obtained by employing the Scatter/Gather approach increases with the increasing number of processors, but the speed difference we get lessens with every added processor. Gain depends on the proportional size of sequential and parallel parts of the document. Gain increases only to a specific upper limit. A certain number of processors will yield maximal possible gain and using additional processors does not bring any extra advantage. The exact mathematical expressions for these conclusions can be found in Appendix A.

Summary of features of SDMP-Scatter/Gather approach:

Advantages:
- faster than SDSP and SDMP-pipeline
- better scalability

Disadvantages:
- work needed to assembly whole system
- not suitable for documents with small percentage of parallelizable content

2.5 MDSP – Multiple Documents, Single Processor

Example: Uploading group of files on FTP server

The substance of this scenario is deciding how to handle the processing of multiple documents in a single processor system. There are two types of MDSP scenarios. Bulk aware MDSP and bulk unaware MDSP.

When working with processing of multiple documents, we will use the concept of an 'average document'. An average document is a representative document with an average size and average properties. The total size of the average documents would have the same total size and same properties as the whole collection of actual files. In the following documentation, when we refer to processing a document, we'll mean an 'average document', unless we state otherwise.

2.5.1 MDSP – bulk unaware

This is a scenario where one SDSP node processes files, which are delivered to it in sequence. It has no knowledge about the relationship between the files that are passed to it. It is a common case in personal computing that a task consists of individual sub-tasks grouped in a batch. An example would be batch file conversion from one format to another.

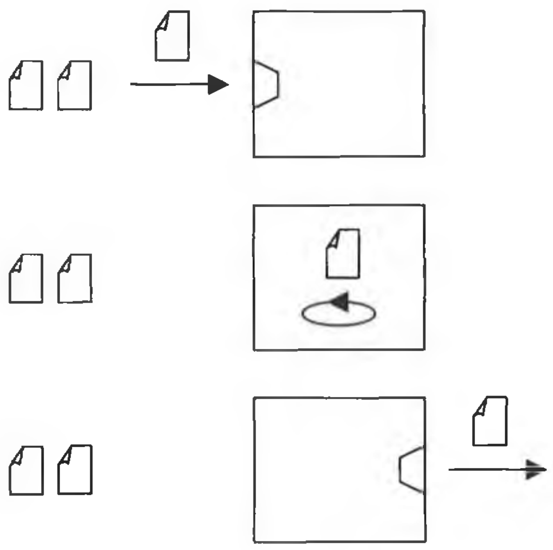


Fig. 2.5 MDSP – bulk unaware scenario

The MDSP-bulk unaware scenario is only a simple SDSP node processing documents one after each other and there is really no time saving that could arise from processing documents in a batch.

Execution time is sum of the completion times of the transformations of the individual documents. The only secondary time savings might come from having some data saved in cache, as they were processed very recently. However, this is dependent on underlying structure and may not happen at all.

Summary of features of MDSP-bulk unaware approach:

Advantages:	-	easy to create
	-	no or very little maintenance coding needed
Disadvantages:	-	no time savings

2.5.2 MDSP – bulk aware

In this scenario, the processing node uses knowledge about the common parts of the document transformations to make the total time of completion minimal. Minimisation can be achieved by reducing the overhead of pre- and post-processing stages which can be run fewer times if they can be shared for more documents in the whole task.

A good example of this approach is uploading files to an FTP server, where pre- and post-processing stage is opening and closing the connection to a server. These stages are run only once and the connection is shared for all files of the batch.

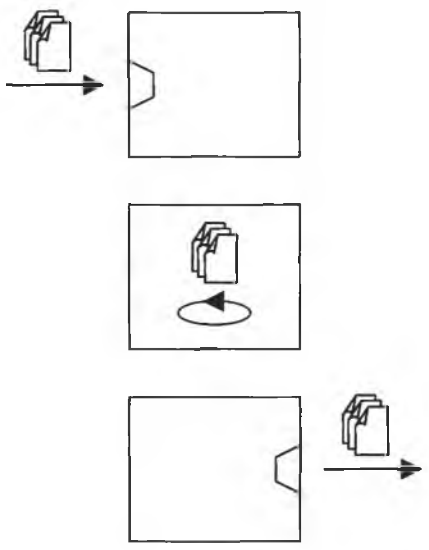


Fig. 2.6 MDSP – bulk aware scenario

In Appendix B we investigate the efficiency of bulk aware processing in detail and reach the following conclusions:

The time saving obtained when using MDSP-bulk aware processing increases with increasing size of the joint section and decreases with increasing size of documents. The percentage of maximal gain obtained by processing a given number of documents is not dependent on function of execution time, but only on number of documents. Exact mathematical expressions for the given conclusions can be found in Appendix B.

Summary of features of MDSP-bulk aware approach:

Advantages:	-	minimizes overhead by sharing parts of transformations common for more documents
Disadvantages:	-	slightly more work to create MDSP-bulk aware system compared to MDSP - bulk unaware
	-	small time savings when joint section of transformations is small compared to length of whole document transformation

2.6 MDMP – Multiple Documents, Multiple Processors

Example: Pipeline multiple-document processing

There are three types of MDMP scenarios. All of these use three groups of processing nodes: a Dispatching group, that cares for the directing of documents in the system; a Transforming group (Transformers), which are nodes carrying out the actual transformation of the documents; and a Collecting group collecting transformed documents or their pieces and putting them together and or transporting them to the specified location so that output of whole MDMP system is produced in the correct manner.

In some systems, some nodes integrate more roles in one. For example, nodes can combine the role of a transforming node and collector, when shipping out the output is an inseparable part of the actual document transformation.

Transforming nodes can be any of systems discussed in the previous sections.

SDSP nodes	–	simple components
SDMP – pipeline nodes	–	more complex pipeline assembled components
SDMP – scatter/gather nodes	–	specialised components for documents with

		parallelizable sections
MDSP – bulk unaware nodes	–	simple bulk processing components
MDSP – bulk aware nodes	–	bulk processing components for documents that have something in common and parts of their transformation processes can be shared

The dispatching unit can take advantage of knowing what the special features of different transforming nodes are and send documents to these nodes that would finish the transformation in the shortest possible time.

All MDMP scenarios look like following:

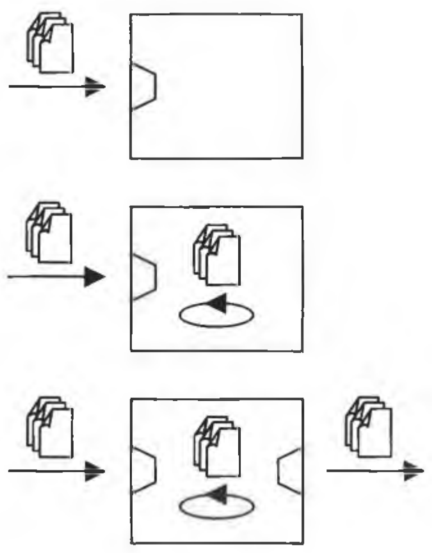


Fig. 2.7 MDMP scenario

The difference from previous scenarios is that documents can flow in and out when other documents are being processed inside the system at the same time.

The MDMP scenario is a pipeline structure, combined with the concept of parallel processing.

The three scenarios differ in how they handle the flow of documents and their dispatching.

2.6.1 MDMP – d-t-c

D-T-C stands for the three node groups employed in this scenario, a Dispatching unit, Transforming nodes and a Collecting unit. Nodes are organised as following:

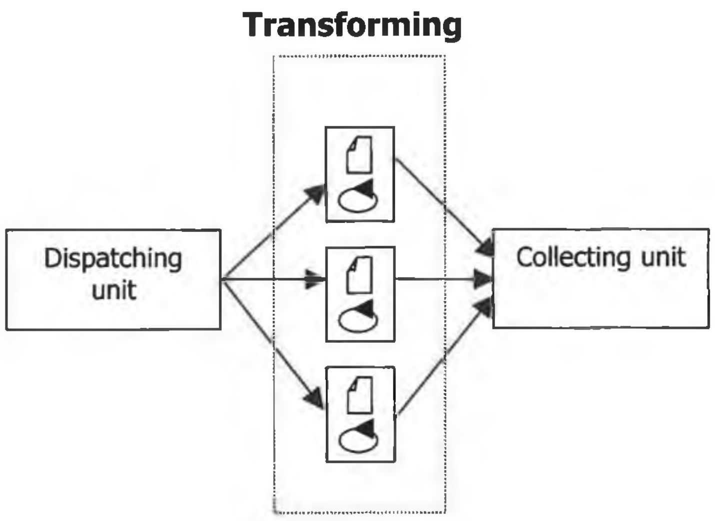


Fig. 2.8 MDMP – d-t-c scenario

It resembles the Scatter/Gather setting and it is indeed the same concept. The only difference is that here it is the individual documents that are sent to the processing nodes, not portions of a document as in Scatter/Gather. It doesn't mean that individual documents can't be processed with Scatter/Gather approach. As the transforming nodes can be any of the previous processing systems they can be SDMP-Scatter/Gather nodes as well.

2.6.2 MDMP – ds-t-cg

A minor variation of the previous approach is ds-t-cg. This has Scatter/Gather functionality shifted to directing nodes. Scatter is incorporated in the Dispatcher and Gather in the Collector.

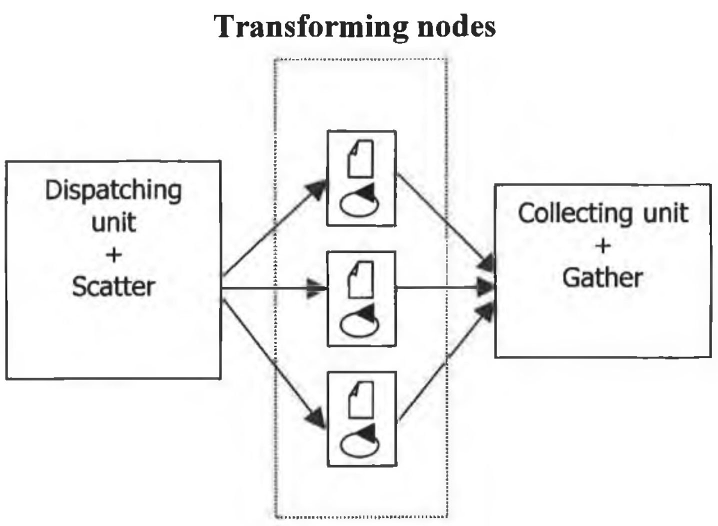


Fig. 2.9 MDMP – ds-t-cg scenario

This change further separates the logic of document maintenance (dispatching, preparing for core-transformation, etc.) and transformation of the document. This allows the transforming nodes to focus more on the transformation itself and not to care about additional issues. The transformation nodes can thus contain less additional logic and be more efficient and, as a result, contribute to an overall reduction in the time of completion.

The dispatching unit has more information about whole job and the state of the system and therefore it can better decide how to handle documents than the individual transforming nodes.

2.6.3 MDMP – pipeline

In the MDMP–pipeline scenario, the document processing system is a pipeline of MDMP – d-t-c or MDMP – ds-t-cg systems. This enables pipelines to contain more documents at the same time. This approach is convenient for large transformations, where documents stay a long time inside the processing system and for transformations of large amounts of documents, that aren't submitted as one batch but rather 'flow in' the system as they come. Furthermore, this system brings a higher level of modularity, which facilitates scalability and thanks to this higher modularity of code it makes monitoring easier.

This approach can be implemented in three different variations.

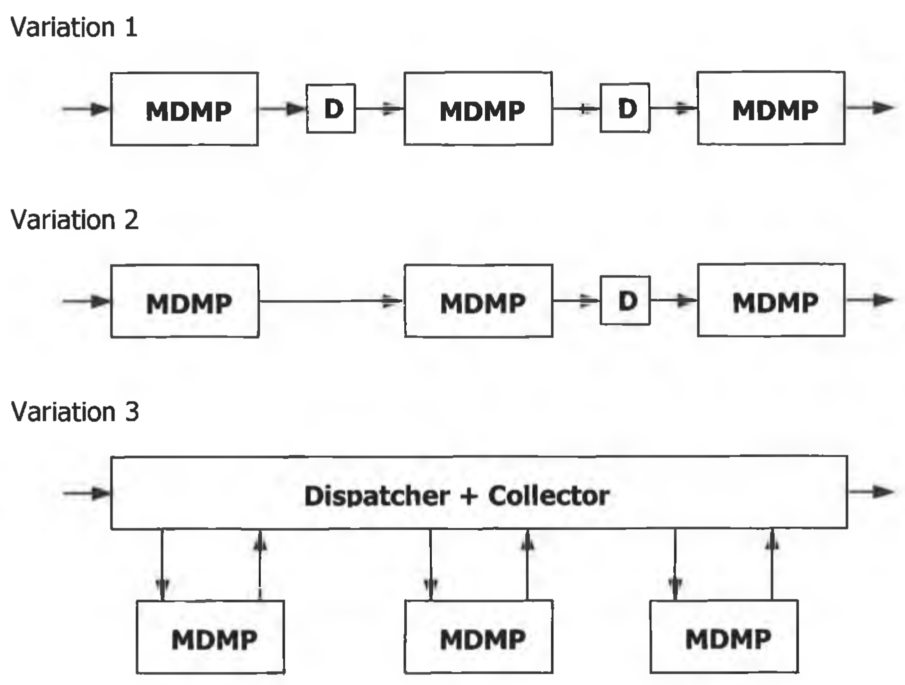


Fig. 2.10 MDMP – pipeline scenario

The first variant has a dispatching unit placed between each MDMP sub-system. The second has dispatching units only after the sub-systems that are substantial in some way and using the dispatching unit in this place brings significant gain. It tries to find a balance between the cost of employing dispatching units and the gain they bring. In a third variation, there are one or more global dispatching units which completely control the flow of documents. In this variant, the dispatcher has most of the information about the state of the whole system and therefore can dispatch documents with the greatest efficiency.

The XPipe paradigm implements the MDMP–pipeline document processing scenario as every component runs different transformations (corresponding to multiple processors) which are carried out on multiple documents entering the pipeline.

2.7 Graphic comparison of document processing scenarios

We can compare different scenarios in regard to their complexity, or how much work we need to do to construct them, versus the speed or their scalability.

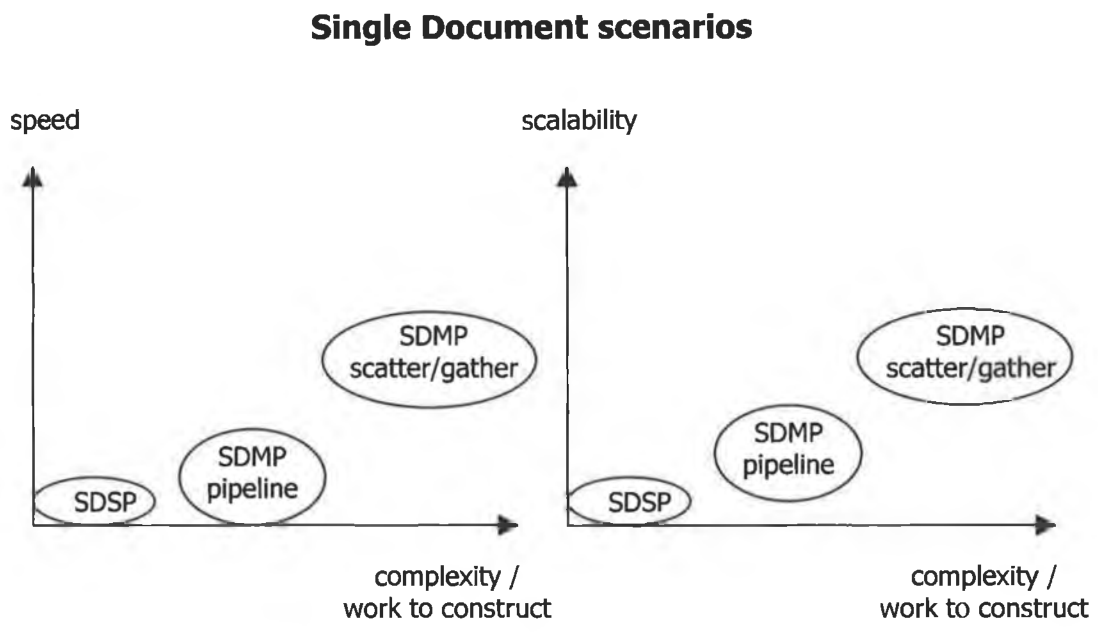


Fig. 2.11 Single Document scenarios comparison

The SDMP–pipeline system requires more work to construct without a gain in speed, but with better scalability compared to the SDSP system. As an additional advantage, it has good re-use capabilities for prospective SDMP–pipeline systems. SDMP–scatter/gather demands even more work than the SDMP–pipeline, but it brings speed and scalability gains.

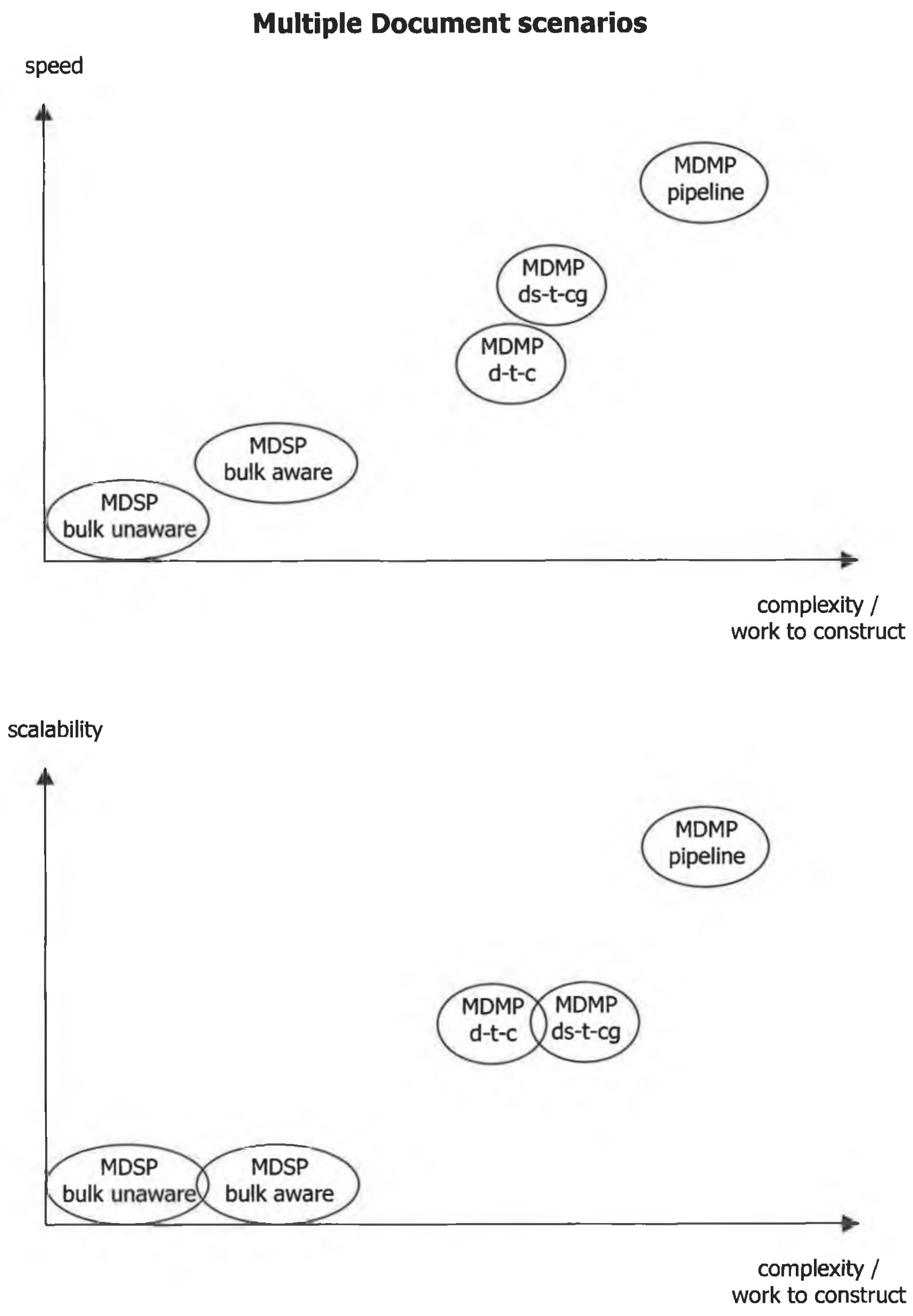


Fig. 2.12 Multiple Document scenarios comparison

MDSP–bulk aware system is slightly more complex than MDSP–bulk unaware, but it brings speed gains by sharing joint sections of the transformation processes. As this is an improvement on MDSP–bulk unaware, rather than a new approach, it does not improve the scalability of the whole system. MDMP – ds-t-cg shows slightly better performance than MDMP – d-t-c because the dispatching logic is shifted to the nodes that have more information about the state of the whole system. As in the previous case, there is no difference in scalability. MDMP – pipeline shows the best performance and scalability, but is the most complex of the multiple document systems too.

2.8 Major concepts of efficient document processing

To close the chapter about the different document processing scenarios we can summarise the techniques that have emerged as being used nowadays in pursuit of increased efficiency of document processing. All these techniques are described in the following list with a brief description of their core idea and features.

1. Bulk processing

Using information or data common to more processed documents.

Benefits: - shorter time of completion,

- less processing power used

2. Pipeline processing

Using smaller code segments / components to execute smaller units of transformations, which are ordered in sequence, passing output of one component as input data to a successive one.

Benefits: - better maintainability

- enhanced monitoring capabilities
- easier load balancing
- support for easy re-use of transformation components

3. Scatter / Gather

Dividing document to smaller segments which can be processed in parallel

Benefits: - shorter time of completion

- easier load balancing

4. Parallel processing

Processing documents on parallel devices and collecting their output to one common location afterwards. In contrast with Scatter / Gather, in parallel processing approach, whole documents are concurrently processed, not only portions of one document as in previous case.

Benefits: - shorter time of completion
- easier load balancing

All these paradigms can be found in previously examined document processing scenarios which leverage their specific features.

CHAPTER 3

CURRENT APPROACHES TO LARGE-SCALE DOCUMENT TRANSFORMATIONS

3 Current approaches to large-scale document transformations

After we inspected the different document processing scenarios, we can focus on one which is of particular interest to us. The focus of this research is on those techniques relating to PropelXbi, which belongs to the MDMP – pipeline document processing scenario.

The objective is to examine how large-scale document processing is currently done and inspect the different areas of the computer world to identify techniques that can be used to enhance PropelXbi as a large-scale document transformation system implementation XPipe.

In the following sections, we give an overview of current state-of-art large-scale document processing techniques. First, in 3.1 we present the straightforward methods being used, then in 3.2 we describe the more advanced approaches and in 3.3 we point out how XPipe relates to the inspected technologies.

3.1 Straightforward methods

The straightforward approach to document transformation is to use one of the technologies available today and compose a transformation. We present five major transformation technologies publicly available today.

3.1.1 SAX

The principle of SAX (Simple API for XML) (SAX) is that a parser processes an input XML file and informs us about the events that happen when it is doing so. Events are of the types: data element start reached, data element end reached, comment reached, etc. The user can define what happens when a particular event is fired. The event calls contain various information, such as the contents of an element, namespace etc. and the user can use them as they prefer.

As the user implements actions only for events that interest him, he does not have to care about processing the rest of the document. This approach also leads to fast execution of the transformation and low memory requirements, as there is no need to allocate large space to save entire document in memory.

SAX technology is a low-level technology and its simplicity is also the source of its disadvantages. It does not provide the user with the functionality of other techniques that facilitate working with elements. Thus, when more complicated transformations are required, too much coding is necessary to simulate the functionality which is commonly available in other higher-level techniques.

3.1.2 DOM

DOM (Document Object Model) (W3C) is a standardized object-oriented model of a document. Parsers providing a DOM interface load the whole document into memory and create a tree model representing the structure of the document. The application can then traverse this document model and work with its nodes representing the document elements. Every node contains information about the element it represents.

By creating a tree representation of document, DOM technology takes the burden of low-level text processing off the developer and allows him to focus on the element transformations. However, this has its disadvantages. As the whole document is parsed and its tree representation allocated in memory, it has high demands on memory space and, if only small transformation is needed or the document is instantly mapped to non-tree model, it creates an unnecessary waste of resources.

High demands on available memory condemns this technology to being unsuitable for large documents, especially if they are going through a sequence of transformations where the whole document tree would have to be recreated in memory again and again.

3.1.3 XSLT

XSLT (Extensible Style Language - Transformations) (W3C 1999) is a language used for converting XML documents by applying templates specifying what transformation should be applied on a particular set of elements.

A template rule has a pattern specifying trees it applies to and an output template used when the pattern is matched. The file processor creates a tree representation of the XML file in memory and successively scans each sub-tree. As each tree in the XML document is read, the processor compares it with the pattern of each template rule in the style sheet. When the processor finds a tree that matches a template rule's pattern, it outputs the rule's template. Templates can perform calculations, can copy content from the original XML document and can work with the initial content as it wants. It can also change the way in which the sub-trees are scanned.

The disadvantage of this approach is an even higher demand on memory space, as three trees are created in memory, compared with two in the DOM case. These three trees are the original document tree, the tree of the XSLT templates and the resulting document tree. This higher demand on memory results in a longer processing time, as more objects need to be created in memory and need to be operated upon.

Another problem for the developer of complex transformations is the complex programming model, which turns out to be cumbersome for realising large-scale transformations. XSLT was never meant to be general purpose XML transformation technology (W3C 1999) and it turns out to be most suitable for simple XML to XML transformations. Even with these weaknesses, it is currently used in some commercial applications performing XML document transformations (BEA; D.I.B.; DataConcert).

3.1.4 DSSSL

DSSSL (Document Style Semantics and Specification Language) is an International Standard for specifying document transformation and formatting in a platform- and vendor-neutral manner. In particular, it can be used to specify the presentation of documents marked up according to Standard Generalized Markup Language (SGML) (Bosak 1996). DSSSL came from publishing community and is widely adopted there. However, as DSSSL aims at transforming complex SGML documents it is complex as well and seemed unsuitable for on-line transformations in Web browser environment. For that purpose a downsized application profile called DSSSL Online (DSSSL-O) was created. DSSSL-O is a profile of DSSSL which removes some functionality and adds capabilities to make it more suited for online documentation. (Quin 2004)

As XML emerged (being simpler subset of SGML) a new simpler style-sheet language was requested and XSLT was constructed, largely based on DSSSL. Both languages share the same concept of application of templates to the trees of elements. Yet, DSSSL doesn't use XML syntax to define transformation templates and as it was more-general predecessor of XSLT, it has more capabilities than XSLT. In contrast with DSSSL, XSLT was widely adopted, whereas DSSSL stayed largely only in the publishing community.

3.1.5 Traditional code – Java, Perl, Python ...

All four technologies mentioned above are focused on XML to XML or SGML to SGML transformations. For transforming documents not in SGML/XML format, traditional code programs are needed. They can process SGML/XML documents as well, but in comparison with SGML/XML-oriented technologies they lose the convenience of document syntactic and semantic pre-processing done by default.

As traditional code is not meant to be a document-transforming technology, it is difficult and inconvenient to create more complex document transformations in it. However, it is useful for converting non-SGML/non-XML documents to SGML/XML for further processing with SGML/XML-focused technologies.

3.2 Advanced methods

When there is need to process either a large number of documents, perform complicated transformation or process documents of large size, the straightforward methods are shown to be insufficient. They are not optimal either in performance or in discriminating complexity of code, which renders the need for more advanced approaches. The following methods use the available technologies in a compound way to achieve the desired processing performance.

3.2.1 Compound monolithic transformations

In the monolithic transformation approach, one object, which conducts the entire transformation, is generated and used afterwards every time a transformation is requested.

Even though the monolithic transformation approach is relatively easy to implement, it has many flaws. As the transformation is performed in one block of code, it is not fault tolerant, and when one portion of the code fails, it infringes all the transformation. Another fault is that systems implementing the monolithic transformation approach don't scale, as there is no way to distribute the execution of a compact block of code. Its massiveness does not allow parallelization either, so only one document can be processed by a system at one time (which is one of the reasons why it does not scale). Finally, for the same reason – the massive nature of code - it is very difficult to monitor and diagnose the execution of transformations, which is a crucial feature necessary when performing the transformations of great numbers of large documents in systems whose execution takes a long time.

The following three methods use the technique of monolithic transformation. These three methods differ both in how the transformation object is constructed and what technology it is built on.

Monolithic transformation – sequence of calls

This first method generates a set of objects, which implement the individual steps of the transformation. The technology used in these objects is not restricted as long as they are executable and can process the given document. It is up to the programmer, or the generator, what code he writes in there. All these transformation classes are then assembled in one embracing script, which successively calls individual transformation objects.

Monolithic transformation – rule based mapping

In this method, the transforming code is generated from a set of business rules describing the difference between the source and target document (Innovations Softwaretechnologie GmbH 2004). Business rules are generally of the form IF *condition* THEN *action*. The condition is usually a match on an element of the input file and the action is a transformation that should be carried out. An example of such a business rule would be: IF source.invoice.invoice_number THEN target.record.add(invoice_number) , stating that when the invoice number is found in the source document, a new record with the invoice number should be written in the target document. In some implementations, this model is extended by the possibility to match on the structure of the target document too and the analyst can then construct the target document from a view of what is yet needed to add to the target document, rather than what is still available in the source document.

When properly implemented, the business rules model can produce efficient transformation code. However, as the rule-based model corresponds to rule inference programming, it is often un-natural to common thinking of programmers and transformation analysts and in many cases sequential programming is a more natural and suitable way of implementing transformations.

Monolithic transformation – semantics based mapping

This method, tries to approach XML data in a different way. XML by itself captures the structure of the data, but does not store any information about the meaning of the data it contains – about the semantics of the data. Semantics, or "ontology", is formal definition of relations between terms (document elements). Such a relation could be "Invoice report is subclass of Report", "Invoice contains invoice rows", "Invoice number is unique integer identifier greater than 1000 and lesser than 10000".

For some particular types of documents, there exist business standards, which define a fixed vocabulary (names and meaning of document elements) used in adhering documents (Commerce One; cXML; ebXML; SAP; OAG; OASIS;

FIX Protocol; ISDA; ISO; RosettaNet). By fixing the vocabulary, they in fact define the semantics (meaning) of the document elements. As a result, when we need to transform documents between two business formats, we can use knowledge about their semantics and automatically create mapping transformations (Contivo 2001). For example, if we had an EDIFACT invoice document and needed to transform it to a SAP IDoc invoice message, we could leverage knowledge of their semantics defined in their specification and generate the transformation which maps the corresponding elements to each other. However, often standards aren't fully overlapping and often one must manually specify the mapping of elements not covered in both standards.

The discussed semantic approach builds on existence of standards for particular types of documents. As a result, use of this approach is narrowly restricted to areas where such standards exist (e.g. exchange of trading messages in financial sector, exchange of standard business messages like invoice, payment, etc.).

Another approach, based on semantic viewing of documents is that of hub-and-spoke. In this approach, one global information model is created which captures all the semantic structure of areas in which we want to carry out conversions (e.g. global information model can be created inside a company which captures a view of all its assets, comprising a view of technical and business personnel). After the global information model (a hub format) is created, to transform, the document in format A to format B, we need to create semantic mappings from both formats to the hub format. As the hub format should capture all the semantics of the transformed documents, transformation form A to B via the common format should be generated mostly automatically. (Contivo 2001; Fox 2003; Unicorn)

This approach appears useful, when a lot of different documents in differing formats need to be viewed in a unified view and various transformations need to be done between them. However, for straight transformation between two formats, (as is case of XPipe and PropelXbi), conversion to a common hub format would create unnecessary overhead.

3.2.2 Pipeline transformations

Several companies implement the pipeline approach, e.g. (iWay Software; Karora; Xbeans). In essence, pipeline consists of transformation components which are chained together into a pipeline and documents pass from one component to another.

Implementations are by and large written in Java using either standard Java or J2EE architecture. Communication systems are mostly built on messaging using JMS or SOAP format. Few implementations use JavaSpaces which is Java implementation of "tuple space"[1] (Karora).

The biggest flaw of all these pipeline implementations is that they allow use of only one transformation technology, which is mostly the application of the XSLT template sheets. Because of the XSLT used as a transforming technique, they inherently suffer from the bad features of underlying technology, which are high demands on memory space and resulting slow processing.

There are two exceptions, namely PerXML (PerCurrence) and Cocoon (Cocoon), which use a different transforming technology. PerXML is an extension of XSLT aimed at extending the template-based transformation approach to allow more intuitive operations and to be able to match on non-XML documents too. Cocoon is a Publishing Framework, which uses SAX filters as a transformation device, and thus it does not suffer from high memory requirements and slow processing. However, as Cocoon is mainly an XML publishing framework aimed at transforming pages, it is not suitable for general and large-scale document transforming. Even more, it explicitly says in its documentation that it is not suitable for the processing of large documents. (Cocoon)

3.3 Position of XPipe

A common problem of all the straightforward technologies mentioned above, is that they are focused on some particular aspect of transforming (low use of

[1] TupleSpaces are discussed in depth in Section 7.1.

memory, ease of access to document elements, etc.) and therefore they are convenient only in some particular area of document transformation. Real-world cases of document transformation do not consist only of these special scenarios and often contain all of them together.

XPipe described in 1.1 solves this problem by allowing individual components of the pipeline to be written in the language most appropriate for the actual transformation stage. Hence, the traditional code can be used to convert a non-XML document to the XML format, SAX code to perform simple maintenance operations and XSLT component can be used for large summarizing actions at the end. This overcomes the performance problem of the majority of other pipeline approaches by allowing the most appropriate language to be used and retaining the flexibility of pipeline composition at the same time.

As XPipe is a pipeline approach, it enables easy monitoring, load balancing and scalability, which are not provided by the monolithic methods. Apart from that, thanks to its modular approach, it allows for composition of very complicated transformations from simple components which can be easily reused and managed.

CHAPTER 4

PROPELXBI AS A SOLUTION OF COMPLEX PROBLEM

4 PropelXbi as a solution of complex problem

In Chapter 4 we look at a study which aimed at a precise characterisation of the computing-related process of complex problem solving and we then look at XPipe and PropelXbi in the light of results of this study.

4.1 Process of complex problem solving

As part of Caltech Concurrent Computation Program (C3P) run on California Institute of Technology, research was carried out which aimed at the characterisation of the processes of solving complex problems in precise terms. The problem solving processes that were the focus of this research were related to computing. This section draws on results of C3P program and all citations in this section are from (Fox, Williams & Messina 1994), which will not be further indicated.

The aim of this research was to create a formal means of looking at the process of solving of complex problem that are related to computing. Researchers wanted to develop a means that allows us to reason about the stages of the problem-solving process and to be able to set a measure on convenience of the proposed solution for different computer architectures.

The fundamental concept is to view the complex problem solving process as a consecutive mapping between complex systems, where complex system is “a collection of fundamental entities whose static or dynamic properties define a connection scheme between entities.” The concept of consecutive mapping is demonstrated by the following example of Airflow dynamics simulation.

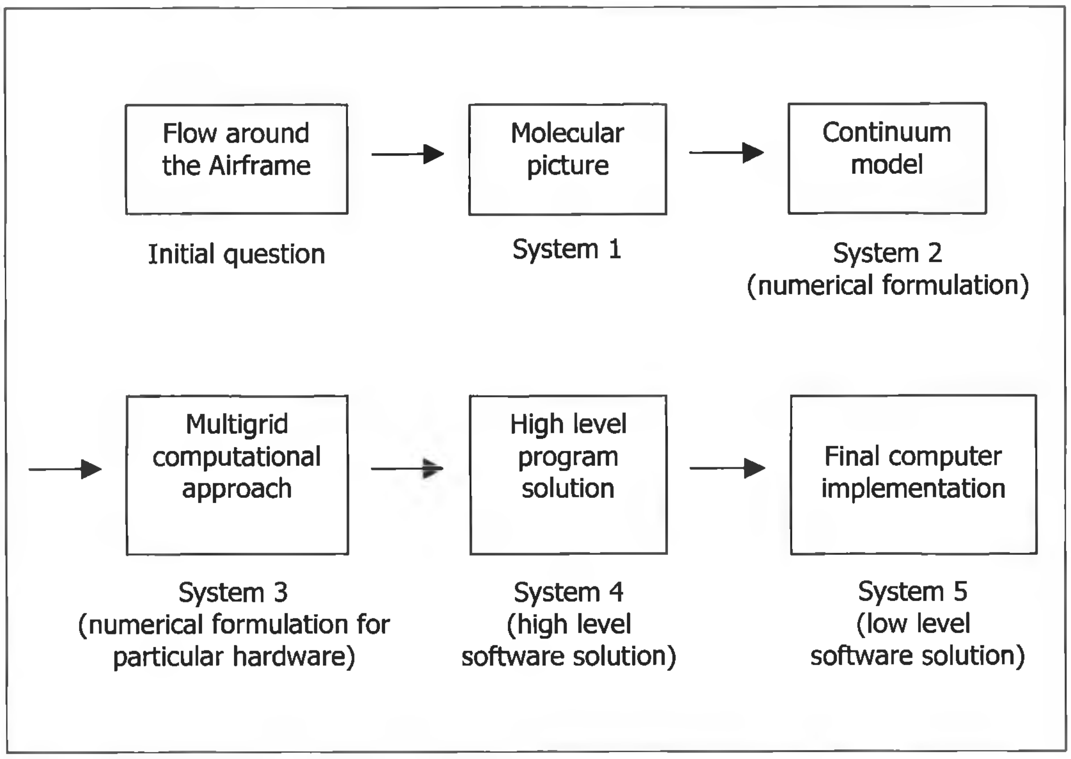

Flow around the Airframe
Initial question
Molecular picture
System 1
Continuum model
System 2 (numerical formulation)
Multigrid computational approach
System 3 (numerical formulation for particular hardware)
High level program solution
System 4 (high level software solution)
Final computer implementation
System 5 (low level software solution)

Fig. 4.1 Airflow simulation problem solution process

The whole problem-solving process starts with an initial task or question, which in this case is a problem of how to simulate the flow of air around an airframe. It is then consequently mapped to the following system representing the logical steps to solving the problem. The last three systems are particularly important as is shown in the following general solving process definition.

Airflow simulation is real-life instance of general problem solving definition, which views process as a sequence of maps between complex systems $S_i, 1 \le i \le k$.

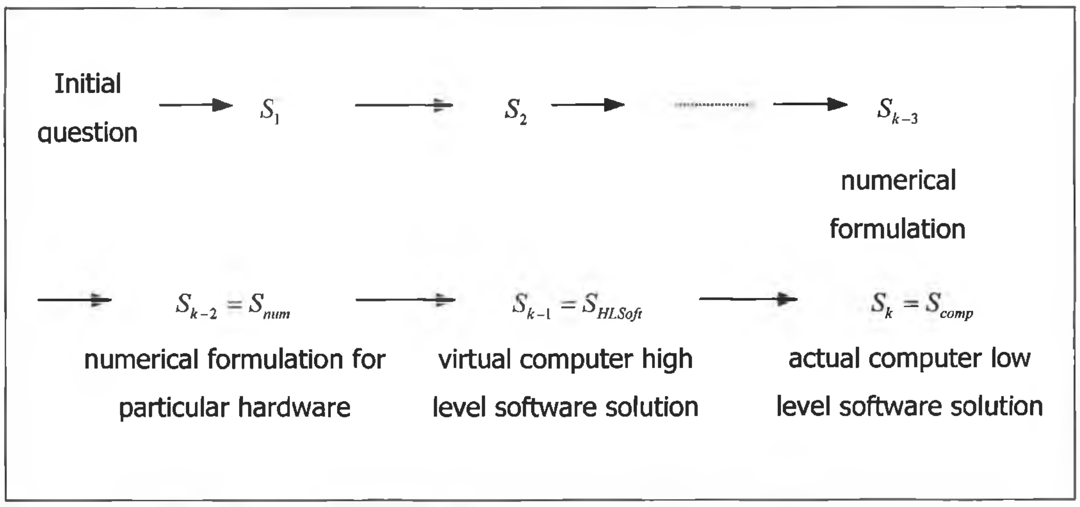

Initial question
S_1
S_2
S_{k-3}
numerical formulation
$S_{k-2} = S_{num}$
numerical formulation for particular hardware
$S_{k-1} = S_{HLSoft}$
virtual computer high level software solution
$S_k = S_{comp}$
actual computer low level software solution

Fig. 4.2 General complex problem solution process

The last three systems have special importance. S_{num} is the numerical formulation of the problem solution with respect to the hardware that is expected to be available. E.g. the numerical solution can take into account the parallel capabilities of the foreseen hardware and may chose a different calculation technique suitable for parallelization. S_{HLSoft} is high level software solution, not concerned about details of computer hardware. The last system S_{comp} represents the low-level software solution, taking into account all the details of the implementation including the hardware communication and issues specific to the chosen architecture.

This general view of the complex problem-solving is important because of two substantial observations:

O-1. (Performance) "Performance of a particular problem or machine can be studied in terms of the match (similarity) between the architectures of the complex systems S_{num} (numerical problem formulation) and S_{comp} (actual computer software implementation)"

O-2. (Implementation appropriateness) "Structure of appropriate parallel software will depend on the broad features of the (similar) architecture of S_{num} and S_{HLSoft}. "

Observation 1 speaks about implementation performance and Observation 2 about the appropriateness of the solution implementation in terms of the matching architectures of complex systems.

O-1 implies that the more distant the implementation is from the proposed numerical formulation (the more original approach is distorted), the worse is the performance of the problem solution.

O-2 can naturally be expected, as the software maps the two complex systems into each other.

One interesting point is that the observations 1 and 2 correspond to the idea of Michael A. Jackson, which says that the structure of programs should correspond to the structure of the input and output data (Sutcliffe 1988). Jackson came to this idea in the 1970's and in the 1980's the more general idea, but with the same fundamental concept, was discovered in parallel computing research by C3P.

4.2 PropelXbi as a complex problem solution

Knowing this high-level general approach to problem solving, we can now have a look at how XPipe and PropelXbi can be viewed as problem solutions of the complex large-scale document transformations problem.

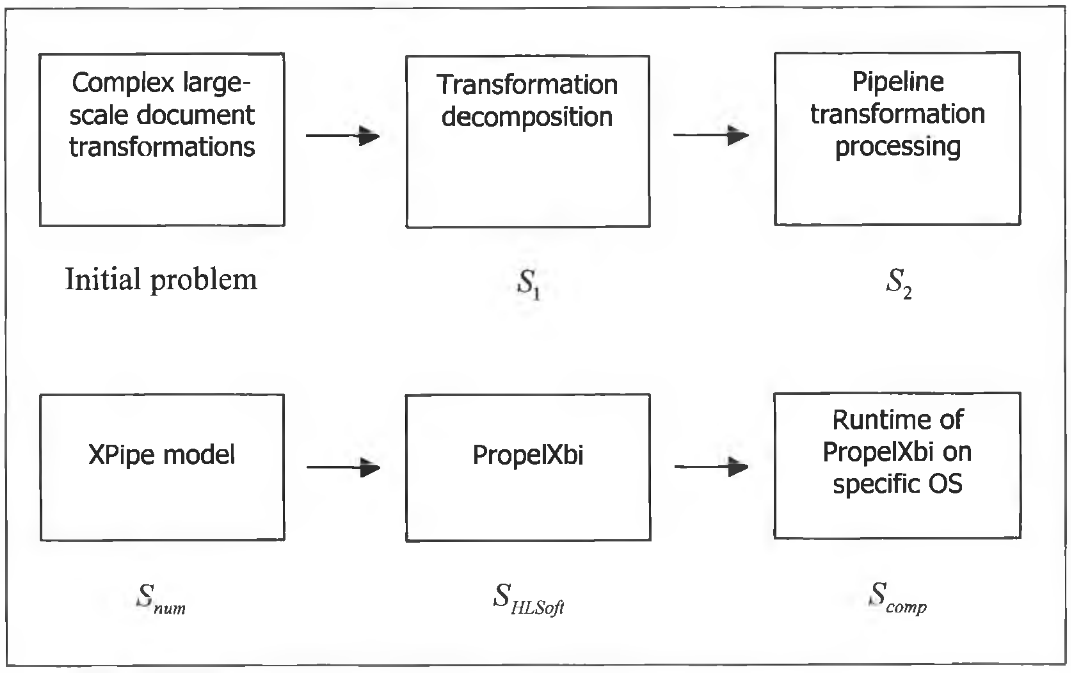

Complex large-scale document transformations
Transformation decomposition
Pipeline transformation processing
Initial problem
S_1
S_2
XPipe model
PropelXbi
Runtime of PropelXbi on specific OS
S_{num}
S_{HLSoft}
S_{comp}

Fig. 4.3 XPipe and PropelXbi as solutions of complex large scale transformations problem

This graph shows consecutive steps leading from stating the initial problem to the actual implementation of a solution. It captures the key steps, which represent the essential thoughts of the XPipe approach – transformation decomposition and pipeline transformation processing which are then embraced in the overall XPipe approach.

The most important asset of this graph is that it identifies what systems S_{num} and S_{HLSoft} are in our large-scale document transformation problem. S_{num} is XPipe and S_{HLSoft} is PropelXbi – an XPipe implementation. This allows us to look at how good implemenation is PropelXbi as a document transformation problem solution as observations O-1 and O-2 refer exactly to S_{num} and S_{HLSoft} (S_{comp} respectively).

As O-1 and O-2 speak about the quality of the problem solution implementation in relation to the match of solution formulation and implementation architectures, the following paragraph compares how well PropelXbi architecture (S_{HLSoft}) matches the architecture of the general XPipe approach (S_{num}).

As XPipe envisages processing as components connected by document queues, the ideal theoretical implementation architecture would be a system of nodes, with one node for each component connected by queues. Ideal nodes would be able to increase its processing power (which is equal to increasing capacity/throughput of queue) in correspondence with the changes of actual workload.

The first realistic implementation architecture would be a data flow computer as it is purposely built to support the flow of data (documents). The problem with this architecture is that it is not realistically realisable, as data-flow computers aren't commercially produced and are not a common part of available hardware facilities. Moreover, the data-flow architecture wouldn't have the ability to increase the computing power of the processing nodes, unless dynamic role assigning could be implemented.

The second best architecture is one involving transputer grids. One transputer node implements one component. It is also connected with other components using a grid. Use of this architecture would have the same problem as the previous one as transputers are not widely used. In fact, transputers are even less common than the data-flow computers today.

The third available implementation architecture is normal personal computers with queues implemented by software pipes. This is in fact a very good and natural match, as pipes are organic parts of computer systems for a long time and are taken as a natural part of the computer environment.

The actual PropelXbi J2EE implementation simulates software pipes as one main queue takes care of passing documents to the appropriate components, as software queues would do. The possibility of dynamically assigning a component to an MDB gives an ability to increase the computing power of the selected component as more allocated MDB's process more documents waiting to be processed by that component.

Another good feature of the PropelXbi implementation is that inter-node communication is built on the JMS message passing system. One of the findings of the C3P project was that "Explicit message passing is still an important software model and in many cases, the only viable approach to high-performance parallel implementations on MIMD machines"(Fox, Williams & Messina 1994). This proves that it was the correct decision to choose message passing as a means of communication in the PropelXbi implementation.

The architecture of PropelXbi matches the general XPipe paradigm and thus fulfils the conditions for good problem solution implementation set by observations O-1 and O-2 in both aspects of appropriateness of implementation and implementation performance.

PART 2

DOCUMENT PROCESSING TECHNIQUES SURVEY

CHAPTER 5

REVIEW OF PARALLEL PROCESSING

5 Review of Parallel Processing

When dealing with parallel XML pipeline processing system, it is useful to look at the different areas of computing where pipeline and parallel paradigms have already been explored. Concepts and techniques used in parallel processing may also be used in PropelXbi and may equally come with a solution of a problem that PropelXbi may face in a future.

First, in section 5.1 we look at architectures which are used in today's parallel computing. Then in 5.2 we discuss which enhancement mechanisms found in the current parallel architectures can be used in enhancing PropelXbi. In the last section, 5.3, we present a classification of parallel problems, state to which particular class document processing belongs and look at how well PropelXbi's architecture matches the problem it is meant to solve.

5.1 Parallel processing architectures

When looking on the area of today's parallel processing, four major computing architectures appear.

5.1.1 Von Neumann architectures

Theses are architectures build on Von Neumann's original concept of a computer. This category includes by far most of the current parallel architectures, comprising SIMD and all the various flavours of the MIMD architectures.

SIMD stands for Single Instruction, Multiple Data. In SIMD architecture, there are multiple processing units performing the same instructions, each capable of fetching and manipulating its own data. An example of SIMD architectures would be vector computers, where whole vectors of data are processed at the same time.

MIMD are Multiple Instruction, Multiple Data architectures. In MIMD, each processor executes its own instruction stream in its unique data stream. Today, nearly all parallel machines are built on MIMD architecture. There are different variations of MIMD, based on a way in which processors access memory. In Shared memory MIMD's all processors have access to pool of shared memory. If there is one level of memory, it is called UMA (Uniform Memory Access) Shared memory MIMD. When there are hierarchies of memory and thus access to different parts of memory can take different time, it's called NUMA (Non-uniform Memory Access). In contrast with Shared memory approach, there are also Distributed Memory MIMD's where every processor has its own local memory and exchange of data is achieved by message passing. Computing clusters and Massively Parallel Processors belong to the later category. (Kaiser; Le & Huu 1997; Plachy 1997; Dongarra 2003; Voicu 2004)

Various enhancement techniques used in parallel computers based on von Neumann concept, which have potential of improving performance of PropelXbi are discussed in later section 5.2.

5.1.2 Dataflow architectures

Significantly different to the architectures based on the Von Neumann concept are dataflow architectures (Duncan 1990). In dataflow computers, computation is driven by the data being processed. Dataflow architectures consist of independent processing units performing fixed operations, which are activated by the arrival of the data to process. After processing the given data, they forward the result to one or more of the processing units.

PropelXbi's architecture is somewhat similar to dataflow, as documents flow from one component to another. However, in PropelXbi, different MDB's can carry out different transformations (depending on what particular XComponent they execute at the moment). This flexibility allows MDB's to be used for whatever transformation is needed and avoids idle waiting which is present in dataflow computers where each processing unit has a fixed function.

5.1.3 Systolic arrays

Another parallel architecture is Systolic arrays (Duncan 1990). A systolic array is a "network of small computing elements connected in a regular grid. All the elements are controlled by a global clock. On each cycle, an element reads a piece of data from one of its neighbours, performs a simple operation, and prepares a value to be written to a neighbour on the next step" (CSEP 1995).

An important feature of systolic arrays is that each processing step is performed in a fixed period of one tick of a global clock, which makes the systolic arrays fast and predictable. Nonetheless, the requirement to have the processing done in a fixed amount of time is not a realistic one for pipeline document processing, as processing in each component can take a different amount of time. Granularity of the steps of document processing is much larger than the granularity of processing in systolic arrays and thus it is not realistically possible to fix the time per processing component.

5.1.4 Neural Network architectures

Neural networks can be viewed as another parallel architecture, as each neuron works in parallel with regard to other components of neural network. However, neural networks are aimed at different work to document processing (e.g. pattern recognition or automatic clustering of complex data), and they are not relative to our case.

5.2 Techniques exploitable in PropelXbi

When researching techniques used in parallel computers based on the Von Neumann concept, the following enhancement techniques emerged. Most were employed in PropelXbi already. As PropelXbi has its specific requirements and architecture, techniques usually had to be adjusted so that they fit the PropelXbi particular case and some were even further enhanced utilizing features particular to XML pipeline processing system.

5.2.1 Pipeline processing

The concept of pipeline processing was first used in 1960's in the construction of computer instruction processing units. Instruction processing was speeded-up as individual machine operations were executed in parallel.

Instruction processing is divided into stages, which can be run in parallel and for every stage there is one (or more) dedicated independent execution units. All these units work with the same clock cycle and pass its output to input of unit realizing following stage. As stages are independent of each other, with every clock cycle, new instruction can start to be processed and after the time of processing one instruction in all stages (called latency), with every clock cycle one instruction is processed.

Ibbett and Topham (Ibbett & Topham 1989) provide equations expressing amount of possible speed-up when using multi-stage pipelines. If τ is the time of a clock cycle (or latency of one stage) and k is number of stages in the pipeline, then the time to process n instructions is $T_k = k\tau + (n-1)\tau$. If we assume that a non-pipelined implementation would take time $T_1 = kn\tau$, what in fact is a pipeline with one stage, then we can express the speed-up resulting from use of k-staged pipe as:

$$S = \frac{T_1}{T_k} = \frac{kn}{k+(n-1)}$$

With higher number of processed documents, speed-up increases with a limit of $\lim_{n\to\infty} S = k$ which is theoretically reached when the number of documents goes to infinity. This limit can never be reached, as the number of documents passing through the pipeline is always finite.

The motivation for this technique was the increase of execution speed. In PropelXbi, though, motivation for using the pipeline approach was different because of the different underlying architecture. Computer processors have a separate dedicated unit for each stage of the pipeline that runs in parallel. In the case of PropelXbi, there is in reality only one (or a not large number) of processors. Because of this limitation, employing the pipeline approach on a single-processor machine doesn't bring any speed-up (as there is nothing that runs in parallel).

In PropelXbi, the rationale for using the pipeline paradigm was to simplify complex document transformations and encapsulate individual simple transformations in the pipeline stages. This decomposition allows for better re-use, easier monitoring and easier maintenance of components working as the building blocks of the whole complex transformation.

The concept of the pipeline is even furthermore extended in PropelXbi, where for every stage a group of nodes is allocated which can be dynamically added or recalled depending on the actual workload in that particular stage. By that, balance can be reached in utilization of computing power of the underlying processor architecture.

If we return to the approach when one pipe stage corresponds to one independent processing unit it would lead us to the thought of using transputers. Every transputer node or group of nodes would be used for one pipe stage and the results would be passed to neighbouring nodes realizing the succeeding stage. In this architecture it would be possible to reach real parallelism of the transformation and achieve speed-up in the same manner as in computer processors. However, the days of transputers are now gone and therefore we have to give up this architecture and focus on implementations achievable on machines that are available today.

5.2.2 Instruction Cache and Instruction Pre-fetch

Both these concepts come from the principle called ‘Principle of Locality’ or ‘Locality of References’ (Liu, Weng & Sun 2001; Prabhu 2003). It says, that relation of instructions of code is linearly related to its mutual locations. In other words, the closer instructions are to each other, the higher is the probability that there is relation between them. This observation results from the nature of programs, which are predominantly sequential and instructions are executed successively as they are written in code.

The concept of a cache comes from two sources. First is that loading from hard disk or any another permanent storage space is expensive in relation to spent time, when on the other hand, time of accessing the same information in memory is many times faster. The second source is the notion that if one address (or file, component) was accessed recently, it is very probable that it will be accessed soon after. This is the implication of Locality of Code because when related instructions have close locality they are executed in near time points as a consequence of sequential execution.

Due to these two reasons, it is beneficial to keep recently accessed items in memory to save the time of loading them from storage space when they are requested again.

The second concept of instruction pre-fetch deals also with saving of loading time. It exploits the direct implication of Locality of Code concept in a way that when one instruction is loaded, processing unit loads several successive instructions ahead of execution. This way, when the next instruction is requested it is loaded in memory already and it is not necessary to wait for its loading. This approach is called look-ahead or speculative execution.

The problem with this approach arises when there are branches in the code (as there usually are). In this case, pre-fetching has to be either disabled until it's resolved which branch will be followed or establish a way of prediction of which way will be taken. In the second solution, called speculative execution, the computer estimates how the code will branch. In case of a right prediction, the execution continues without any changes. In the opposite case, all pre-fetched instructions have to be cancelled and instructions of the right branch have to be loaded from scratch. The latter case causes delays in execution and therefore it is crucial to choose a good prediction method.

Both these concepts are implemented in PropelXbi in a modified way taking advantage of PropelXbi particular architecture. XPipe can be likened to normal sequential code, where each XComponent corresponds to one sequential instruction. There are two chief differences from the computer program, though, which allow PropelXbi to implement Cache and Pre-fetch concepts even more effectively than they are implemented in CPUs.

The first, and crucial, difference is that there is limited, and small, number of components and that their size (2 kilobytes on average) is small compared to amount of working memory which is available. In case of computer instruction processing units, the size of the cache was limited to the order of tens of cached instructions which is a minute fraction of the number of instructions in common programs, which is higher by several orders.

The second difference is that there are usually only a few branches (if any) and the sum of XComponents of all possible branches is still small in comparison with the available memory. The situation with PropelXbi is also better by another factor, which is that the XComponents that would be loaded in the wrong branch prediction wouldn't have to be removed from memory and thus time that would be spend by memory erasing would be saved.

Because of these particular characteristics, it is possible to load all the XComponents that will be accessed throughout the XPipe execution into the memory on start-up of the whole XPipe and keep them in memory without any further need to access the storage structure in which they are saved. This way the Cache and Pre-fetch concepts are employed at once with even greater efficiency than they are implemented in architecture of computers.

5.2.3 Data forwarding

The concept of data forwarding originated from the significant ratio of time spent by sending data from execution units to registers and the time of actual function execution in a unit. The time of delivering results to registers usually is not negligible compared to the functional unit execution time and thus when consecutive instructions work with the same elements, they can be directly sent from the first unit to the second, without the need of saving the intermediate result to registers and loading them again.

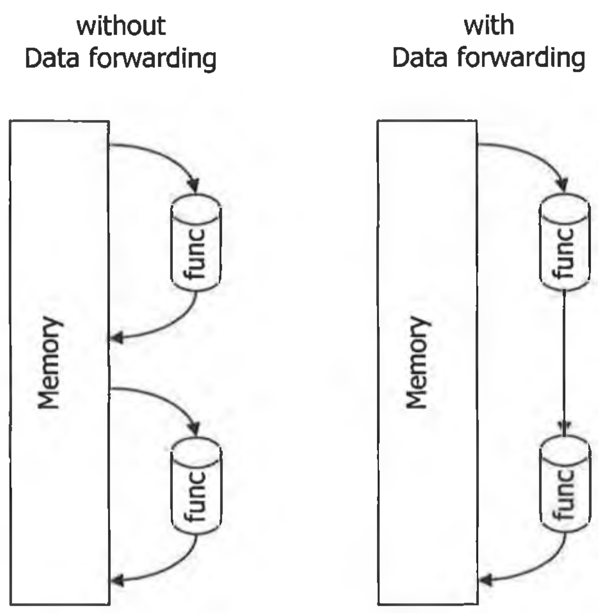


Fig. 5.1 Data forwarding

In XPipe *every* succeeding component works with the same element as previous one and therefore XPipe looks like ideal candidate for employing data forwarding.

A problem arises with the Java implementation of XPipe - PropelXbi, which is based on queues and the dynamic pooling of Message Driven Beans. The PropelXbi architecture was discussed in detail in section 1.2. In brief, there is a queue, which stores XML documents and a pool of working nodes (Message Driven Beans – MDB's) which load these documents from the queue, execute individual stages of the whole document transformation (XComponents) and return the transformed documents back to the queue.

If data forwarding was implemented in PropelXbi, the working node would send the transformed document directly to the following worker if there were any free. If there were not, it would save the document to the queue as normal. Gain would be obtained from the shorter time between the end of executing one stage and the beginning of the next one as the document would not be passed to the queue and loaded to the worker, but would be sent directly to him.

It could be thought that even greater time saving would be achieved if the worker would not pass the XML document to another one, but would process the next transformation stage itself (thus removing the time of passing the XML document to the following worker). However, this approach would not allow the dynamic assigning of working nodes to stages where there is higher workload and monitoring facilities would be degraded as well.

One major problem is that the Message Driven Beans can send messages only to queues and do not allow sending messages to some other particular MDB. It might be possible to implement a pipeline consisting of Stateless Session Beans passing results directly to each other, but it would mean that the facility of dynamic working node assigning would be lost.

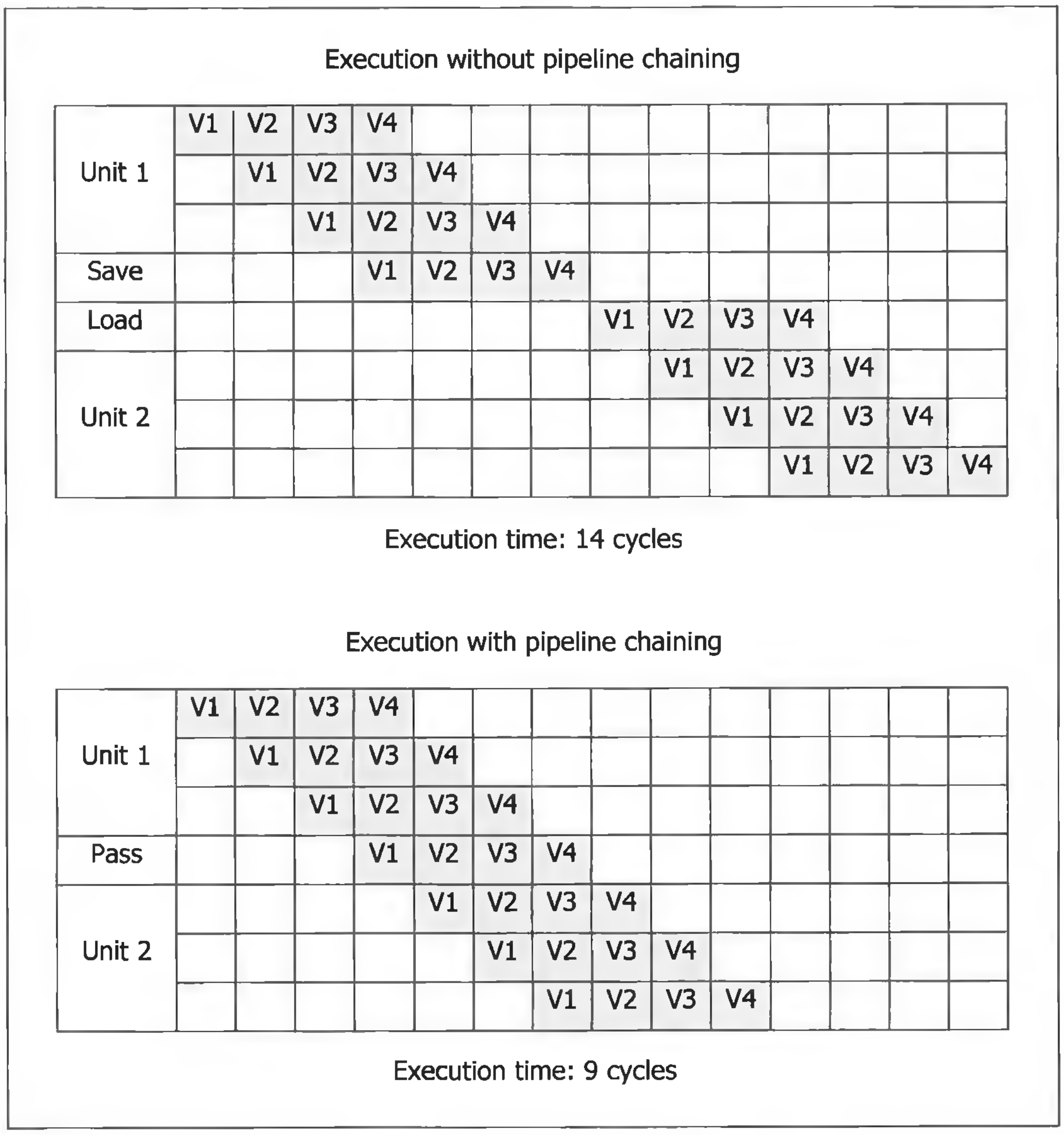
Another fact concerning the data forwarding implementation in PropelXbi is that the time spent by passing documents to and from the queue is negligible compared to how much time documents spend waiting in queues. Because of this disparity, the gain that would be obtained would not be of any significant size.

In light of these facts it appears that employing data forwarding in PropelXbi would not be beneficial because the cost of the loss of dynamic MDB assignment would be greater than the gain obtained by the occasional removing MDB -> Queue -> MDB communication. Nevertheless, the promising concept of data forwarding does not have to be fully abandoned. In fact, it can be exploited in these parts of the pipeline where the loss of the possibility of dynamic assigning does not matter because the execution time of components is small and coupling components together delivers significant gain. These are the cases when the XComponent compiler can be used. Compiled components then represent the extreme case of close coupling of functional units where the time of passing documents from one component to another is very near to zero. The concept of the XComponent compiler is discussed in greater depth in sections 6.3 and 6.4.

5.2.4 Vector pipeline chaining

Vector pipeline chaining is a technique used to speed-up the processing of pipelined functional units of vector computers. On vector computers, one instruction can be used to process the whole vector of data and thus entire block of data is loaded and processed at once. This processing is done in pipelined units and because all vector elements are preloaded, it can start processing the next vector element every clock cycle, even though the execution time of the pipeline is many times longer. When the situation occurs where consecutive vector instruction refers to the result of a previous one, the functional units can link together so that the second unit doesn't wait until the execution of the vector in the previous unit is completely finished, but starts processing vector elements as soon as they reach end of the preceding pipeline. This resembles the data forwarding technique, but the difference is that this case employs the coupling of units on a higher level where data (data vectors in this case) are forwarded even before they are entirely processed.

The effect of pipeline chaining is demonstrated in the following picture, which shows the time diagrams of two 3-stage pipelined units processing a vector of four elements. The horizontal axis indicates the time in clock cycles. The vertical axis indicates in what stage the elements of vector (V1–V4) are located.



Execution without pipeline chaining

Unit 1	V1	V2	V3	V4										
		V1	V2	V3	V4									
			V1	V2	V3	V4								
Save				V1	V2	V3	V4							
Load								V1	V2	V3	V4			
Unit 2									V1	V2	V3	V4		
										V1	V2	V3	V4	
											V1	V2	V3	V4

Execution time: 14 cycles

Execution with pipeline chaining

Unit 1	V1	V2	V3	V4										
		V1	V2	V3	V4									
			V1	V2	V3	V4								
Pass				V1	V2	V3	V4							
Unit 2					V1	V2	V3	V4						
						V1	V2	V3	V4					
							V1	V2	V3	V4				

Execution time: 9 cycles

Fig. 5.2 Vector pipeline chaining

Figure 5.2 clearly shows the advantage of using pipeline chaining. The time saved by employing this technique increases with the size of the vector being processed as the saved time can be expressed as $t_{saved} = n + 1$, where n is length of the vector.

Even though, it may seem that the vector pipeline chaining isn't utilisable in PropelXbi, as there aren't any corresponding vector structures on the document level, it can be used at a lower level, where a document is considered to be group of interrelated elements. When these elements are self-contained, like for example in a document representing an invoice, containing independent elements representing individual invoice rows, these elements can be extracted, producing a set of autonomous documents, which can be processed in parallel independently of the others. Thanks to that, parts of document can be processed by later stages even though some other parts were not yet processed by stages placed earlier in the transformation pipeline.

In PropelXbi, this functionality is implemented by Scatter and Gather components, as described in section 1.1. Scatter divides the document into a set of independent documents, which contain separated elements of the original file. After they are all processed, the Gather component assembles them into the resultant document.

5.3 Parallel problem classes

Apart from the categorisation of different parallel architectures, the literature about parallel computing also suggested the division of problems, which are solved by parallel computing and recommended architectures which match the given problem type the best. This section looks on the categorisation of parallel problems and looks on how the PropelXbi's architecture matches our problem of document processing.

In (Fox, Williams & Messina 1994) Fox et al. categorise problems solved by parallel computing into following five classes.

5.3.1 Synchronous

This class represents tightly coupled problems where the software needs to exploit features of the problem structure to get good performance. Compared to problems of other classes, synchronous problems are relatively easy as different data elements are essentially identical. An example of a synchronous problem is the numerical solution of a magnetic field distribution.

5.3.2 Loosely Synchronous

Loosely synchronous problems have the same nature as synchronous, but with the difference that the data elements are not identical. One of loosely synchronous problems is solving sparse linear algebra equations.

5.3.3 Asynchronous

Asynchronous problems display functional parallelism, which is irregular in space and time. This means, that relations between data elements changes with proceeding time (irregular in time) and that node interconnection needs to change with proceeding time as well to suit the problem (irregular in space). A good example of asynchronous problem is game of chess.

Problems in this class are often loosely coupled and so it's not necessary to worry about optimal decomposition to minimise communication. These problems are usually hard to parallelize, unless they belong to following special class of asynchronous problems.

5.3.4 Embarrassingly Parallel

Embarrassingly parallel problems are type of asynchronous problems, where components solving the problem can be executed independently and mutual communication is sparse if not unnecessary at all. These low communication requirements make them particularly suitable for a distributed implementation on a network of workstations. An instance of an embarrassingly parallel problem is the radio signal frequency analysis performed by SETI@Home project (SETI@Home).

5.3.5 Compound Metaproblems

Metaproblems are asynchronous collections of asynchronous, synchronous or loosely synchronous components where these programs themselves can be parallelized. An example of compound Metaproblem would be army tactics decision support software.

The case of large-scale document transformation problem handled by PropelXbi belongs to class of Embarrassingly Parallel problems. Documents entering PropelXbi have nothing in common (at least form transformation point of view) and so they all can be processed separately and no communication is needed between processing components. The only communication which needs to be carried out is passing documents from one stage to another.

Fox at al. also suggest most suitable ("correct") mappings of a problem to software, machine pair (Fox, Williams & Messina 1994).

Problem Class	Software	Machine
Synchronous	Synchronous	SIMD or MIMD
Loosely Synchronous	Loosely Synchronous	MIMD
Asynchronous	Asynchronous	MIMD (but may not work well without special hardware features)
Embarrassingly Parallel	Asynchronous	Network of MIMD workstations
Compound Metaproblems	Asynchronous with heterogeneous components	Heterogeneous network

Tab. 5.1 Suitable mapping of a problem to software, machine pair

Table 5.1 suggests that embarrassingly parallel problems should be handled by asynchronous software on MIMD architectures. PropelXbi is exactly this case as it implements a system where every component can perform different transformations (MIMD architecture) and communicates by asynchronous communication message-passing system. This correspondence shows that XPipe is a correctly chosen approach for large-scale document transformation.

The project of SETI@Home shows the eventual extension of PropelXbi, where documents would be processed on the same basis as in SETI – processed on computers that would otherwise stay idle. Thus a computer gets whole pipeline and batch of documents to process and processes them when computer's usage is low for a set amount of time. From a theoretical point of view, there is nothing that would stop PropelXbi from expanding in this direction.

CHAPTER 6

JACKSON INVERSION SURVEY

6 Jackson Inversion Survey

In section 4.1 "Process of complex problem solving", we saw that one of findings of C3P parallel computing project was, in other words, that the quality of software is related to how much it reflects the data it works with. Interestingly, a corresponding idea was pronounced by Michael Jackson ten years earlier in relation to how to write sequential programs efficiently. It appears that this idea has general validity and it is useful to have a look on the work of Michael A. Jackson in more detail, particularly at his principle of Jackson Inversion, which has remarkable similarity with the scenario, which is dealt with in PropelXbi.

In this chapter, we firstly introduce the essential idea of Jackson Structured Programming and Jackson Inversion. In the following section 6.2 we examine how beneficial it would be to employ Jackson Inversion in PropelXbi and in closing section 6.3, we examine the concept of an XComponent compiler, which develops from findings of two previous sections.

6.1 Jackson Inversion and Jackson Structured Programming

Jackson Inversion was developed by Michael A. Jackson, a computer scientist in the area of information systems development. It is part of broader work called JSP – Jackson Structured Programming, which originated in the 1970's by examining how sequential batch-processing systems were written and how they should be constructed in order to be effective. (Sutcliffe 1988; Ourusoff 2003)

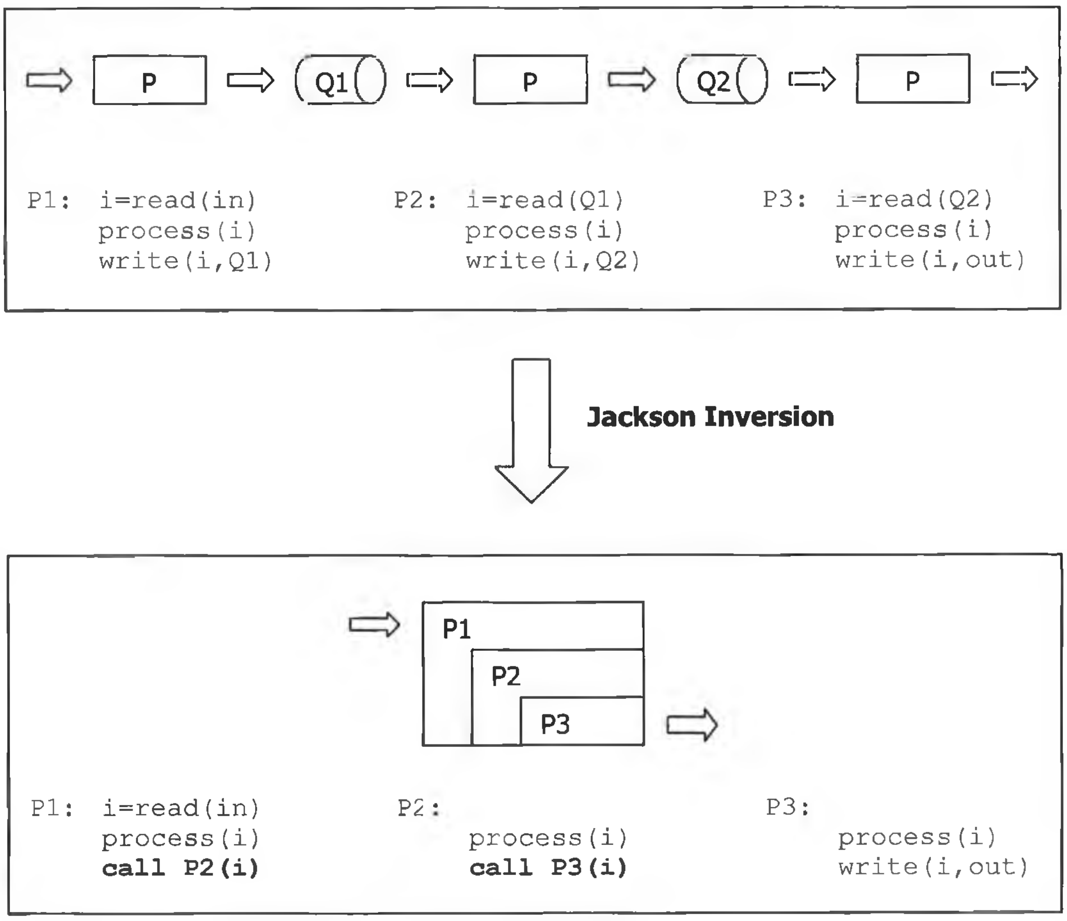
JSP is an approach how to design and implement programs, so that they are effective and easy to modify when system requirements change. The central idea of JSP is to write programs so that structure of code reflects the structure of input and output data.

Jackson Inversion is a method to simplify complex systems of programs communicating with each other through temporal storage spaces or in other words through queues.

The idea itself is to transform original individual programs to one block, where programs call each other directly. By this technique, the need for intermediate queues is eliminated and thus these intermediate parts are removed.

As described here, Jackson Inversion could be also called with the more descriptive word ‘absorption’, as called programs are absorbed into calling code.

Jackson Program Inversion is demonstrated by following picture:



P
Q1
P
Q2
P
P1: i=read(in) process(i) write(i,Q1)
P2: i=read(Q1) process(i) write(i,Q2)
P3: i=read(Q2) process(i) write(i,out)
Jackson Inversion
P1
P2
P3
P1: i=read(in) process(i) call P2(i)
P2: process(i) call P3(i)
P3: process(i) write(i,out)

Fig. 6.1 Jackson Inversion

Even though the picture above suggests that P1, P2 and P3 are all written in one file, they may be different programs merely being able to call each other.

The process of Jackson Inversion is in fact the transformation of asynchronous system to synchronous.

Summary of Jackson Inversion features follows:

Advantages:
- system simplification
- execution speed-up

Disadvantages:
- by removing pipes, we force system to be synchronous and thus loose advantages of asynchronous execution
- not convenient for data-driven applications (limited to sequential batch processing scenarios)
- it is more difficult to add or remove individual functionality components in one big monolithic code than it would be in previous decomposed system
- resulting program is modular, but monolithic code. This goes against whole concept of transformation decomposition into small individual transformation components which is base of entire XPipe paradigm

6.2 Suitability of employing Jackson Inversion in PropelXbi

PropelXbi is in its nature a big asynchronous pipeline containing large number of components. The problem of employing Jackson Inversion in PropelXbi arises from its transformation from an asynchronous, event driven system to a synchronous, code driven system. This transformation causes loss of processing efficiency.

In PropelXbi, document transformation is triggered by arrival of a document to pipeline. In an inverted system, transformation is initiated by the first component of the pipeline. If there isn't any document to process, the first component waits until it arrives. Otherwise, it takes a document in, processes it and passes the partially transformed document to the following component. This iterates until the document reaches the last component, which finishes the transformation and outputs the final document. The last component then finishes its running and processing is returned to the previous component, which repeats until the first component is reached. If there isn't any document to process, the first component waits for arrival of such data. Otherwise, it processes document on input and whole transformation runs again.

Difference in processing is shown on following process diagrams.

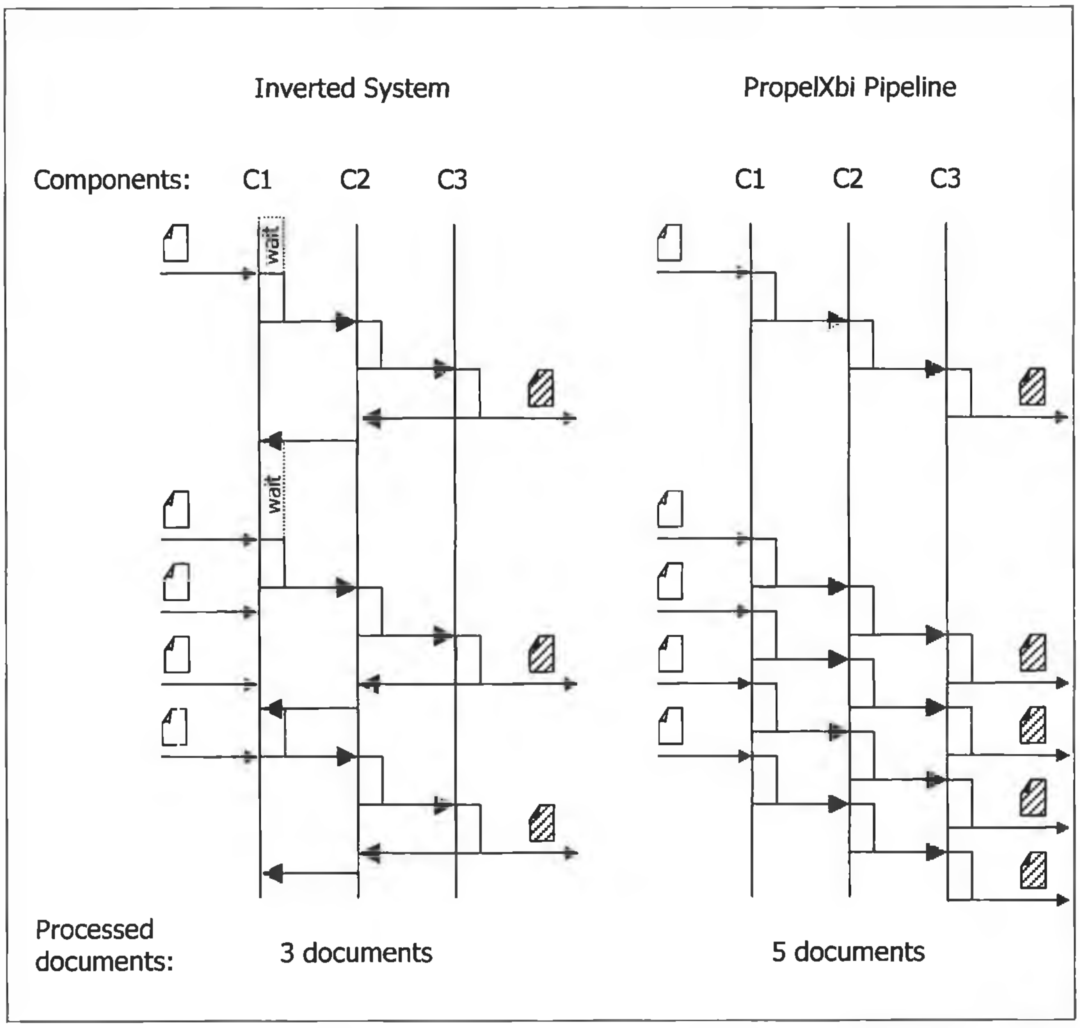


Fig. 6.2 Comparison of processing in Inverted system and Pipeline

This figure demonstrates two problems rising when employing Jackson Inversion

1. Components are blocking each other (problem of synchronicity)
2. Redundant use of processing time (need for passing call from last to first component)

A disadvantage not displayed on the picture above is that it is not possible to use Scatter/Gather in monolithic inverted program. The advantage of Scatter/Gather comes from independent parallel processing of portions of document which isn't possible in serial code produced by Jackson Inversion.

There is an improvement based on the Scatter/Gather method which could be used to increase the efficiency of Inverted code. It can be only applied when

$$O(t(n)) > O(n) \textbf{ AND } p > 0$$

i.e. order of time function is greater than linear (e.g. quadratic, polynomial ..) and there are portions of document which can be processed separately. n stands for the size of document and p for the portion of document which can be processed in parallel. The idea is to divide the document into as small portions as possible and then process these small portions sequentially one by one.

As $O(t(n)) > O(n)$, the sum of processing times of portions is smaller than the processing time of the whole document.

$$\sum_{portions} t_{portion} \leq t_{whole\_document}$$

To sum up, we can say that it would be beneficial to employ Jackson Inversion in PropelXbi by inverting whole pipeline to Inverted code only if:

1. Gains earned by removed component communication are greater than gains earned by employing parallel processing in Scatter/Gather (improbable)

2. "Condition of Asynchronity Loss Acceptance"
   The time between document arrivals is greater than average time of completion of document transformation (might occur)

$$t_{arrival} > \overline{t_{transformation}}$$

If both these conditions were fulfilled then Jackson Inversion would have the following Pros and Cons:

Advantages:
- execution speed-up

Disadvantages:
- loss of ability to dynamically allocate MDB's depending on actual workload
- processing time used even when there aren't any documents to process (phase of returning call from last to first component)
- no scalability (once pipeline is inverted, it looses scalability)
- difficulties with clustering

The strongly limiting condition (2) of document arrival period greater than transformation period leads to the concept of an On-line and an Off-line XComponent compiler, which are discussed in following sections.

6.3 PropelXbi on-line XComponent compiler

The concept of an on-line XComponent compiler (XCOc) is to compile portions of pipeline in cases when it would bring execution speed-up and the conditions for useful employing of Jackson Inversion would be easily fulfilled.

In order to be able to analyse when it is beneficial to employ an on-line XComponent compiler, we revisit the PropelXbi architecture as described in detail in section 1.2. The PropelXbi architecture consists of main Queue, Working nodes – workers (Message Driven Beans – MDB's) and Executive.

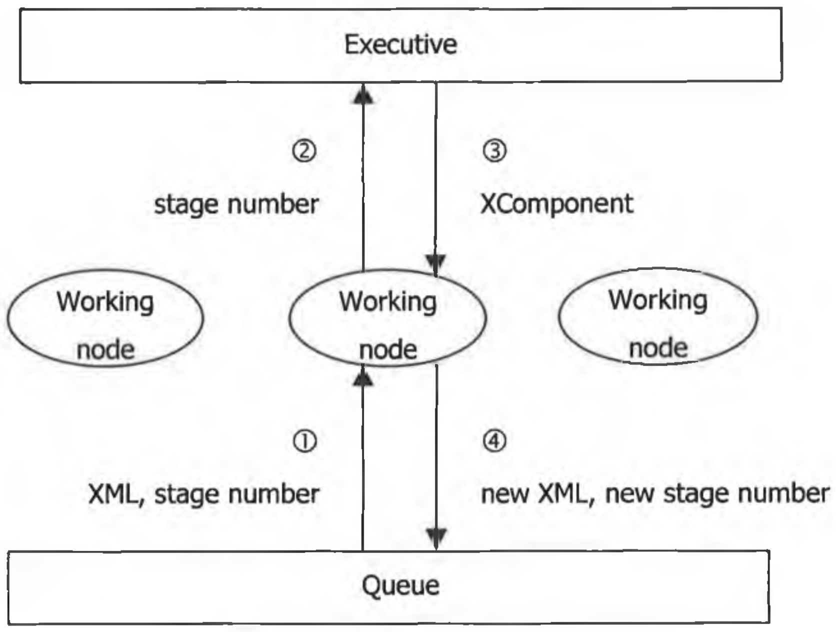


Fig. 6.3 PropelXbi processing architecture

The queue serves as a storage space for documents in their intermediate phases. The working nodes watch the queue and when there is some document to process, they load it in together with information about with what transformation (XComponent) should be used. The Worker then asks Executive for appropriate XComponent and Executive loads it and passes it to the working node. Afterwards the worker executes the transformation and saves new document in storage space together with information about what transformation should be applied to this document next.

The following process diagram shows one of these transformation steps.

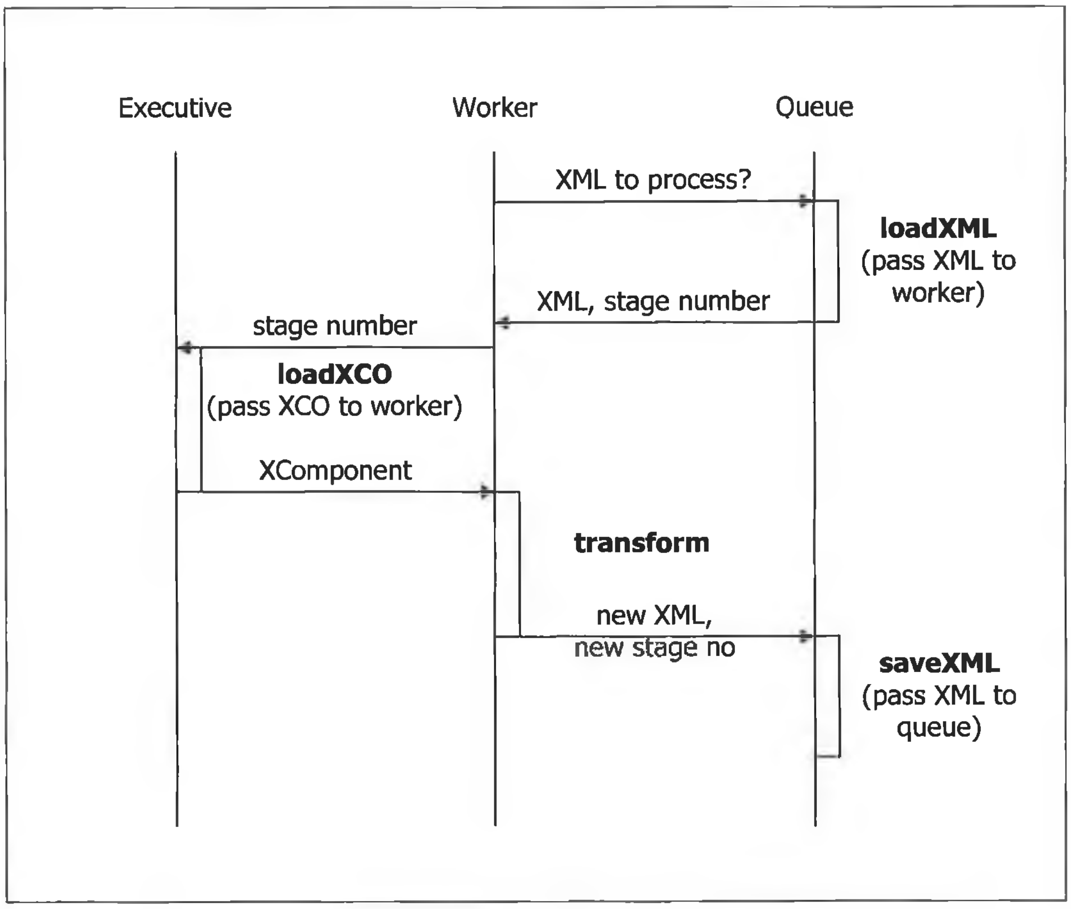


Fig. 6.4 Transformation step process diagram

The condition advising when to use on-line XComponent compiler (XCOc) comes from the observation of how the transformation process changes when XCOc is used. In the following we compare the process sequence of two successive components being executed sequentially and the sequence when these two components are compiled together with XCOc.

without XCOc	**with XCOc**
`loadXML 1`	`loadXML 1`
`loadXCO 1`	`compileXCO 1-2`
`Transform 1`	`loadXCO 1-2`
`saveXML 2`	`Transform 1-2`
`loadXML 2`	
`loadXCO 2`	
`Transform 2`	
`saveXML 3`	`saveXML 3`

For simplicity we assume that the time of `loadXCO 1-2 = loadXCO 1 + loadXCO 2` and `transform 1-2 = transform 1 + tranform 2`. Under this

assumption, the steps that differ in these two processes are `loadXML 2` and `saveXML 2` in the former and `compileXCO 1-2` in later.
These two segments show when the use of on-line XComponent compiler is beneficial. For it to be so, the following condition has to be satisfied:

$$compileXCO1-2 < loadXML2 + saveXML\,2$$

where comparing two segments is considered to be comparing their execution time. In other words, it says that time spent by compiling must be lesser than time spent by extra communication (maintenance).

For general case of $n \geq 2$ components $XCO_1..XCO_n$ inequality generalises to

$$compileXCO1-n < \sum_{i=2}^{n}(loadXML\,i + saveXML\,i)$$

With knowledge of this condition and consideration of "Condition of Asynchronity Loss Acceptance" ($t_{arrival} > \overline{t_{transformation}}$), it is the work of Executive to decide whether to compile certain parts of the pipeline together or leave them in original "monoidic" form.

In order to be able to decide whether to compile XComponents $XCO_1..XCO_n$ the Executive needs to know following information:

- $t_{arrival\,XCO1}$ average document arrival period of first XComponent

and execution times of:

- `compileXCO 1-n`
- `loadXCO 1-n`
- `loadXML i` of all XComponents
- `saveXML i` of all XComponents
- `transform i` of all XComponents

This information can be measured and gathered by Executive in cooperation with the Queue and working nodes.

To incorporate on-line XComponent compiler into the current implementation of PropelXbi, the following steps need to be done:

1. Code on-line XComponent compiler
2. Add logic to Executive to enable it to decide when to use XCOc
3. Modify working nodes (MDB's) so that they set the number of next transformation (XCO) to correct value (increment counter by correct number depending on how many components were applied in transformation they executed)

Note: Code generated by XCOc differs from code that would be obtained by applying Jackson Inversion in the way in which individual components are assembled together. Jackson Inversion produces hierarchy of nested components whereas XCOc would rather construct sequence of components fitted in uniting skeleton.

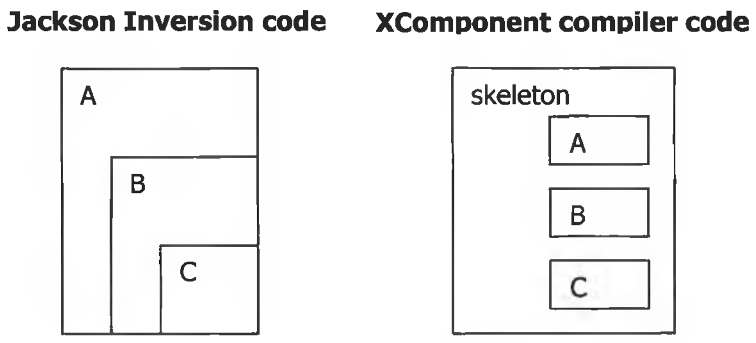


Fig. 6.5 Jackson Inversion and XComponent compiler code

The later approach allows easier exception handling and makes it easier to monitor the transformation process.

6.4 PropelXbi off-line XComponent compiler

The concept of off-line XComponent compiler (XCOc) is to compile XComponents of pipeline into standalone package (compiled pipeline), which can be used to transform documents without need of running whole PropelXbi engine. As XComponents are designed as black boxes, which simply take a document in and output it out, a compiled pipeline can be used as another XComponent as well. The Off-line XComponent compiler differs from the on-line version in that XComponents chosen for compilation are not chosen in execution time by Executive, but by the user, without the need of having PropelXbi running.

The functioning of off-line XCOc is envisaged as following. It takes the name of pipeline to compile as an argument and checks if it is possible to compile all its XComponents into java classes. XComponents that can be transformed in such a way are normal Java classes, XSLT sheets, which can be compiled with XSLTc and Jython scripts, which can be compiled to Java code as well. If it is not possible to compile some XComponents it returns with error, otherwise it compiles all components and generates a handling class with a fixed name (e.g. `transform`) which handles sequential passing of incoming document to one XComponent after another in the order defined in the pipeline description. Apart from document passing, the handling class also caters for exception management, error reporting and correct functioning of Scatter/Gather components. As a final step, off-line XComponent compiler packages all created classes into a jar file with the name of the original pipeline.

A compiled pipeline created in such way then can be used from the command-line with simple command:

```
java -cp MyPipe.jar transform In.xml Out.xml Error.xml
```

Chapter 7

Review of Distributed Computing Technologies

7 Review of Distributed Computing Technologies

As found in previous chapters the problem being solved by XPipe belongs to the class of embarrassingly parallel problems. Problems in this class are ideal candidates for loosely coupled distributed computing solutions and thus it is beneficial to examine some technologies from the present distributed computing world. These technologies could be used for the expansion of PropelXbi into the distributed world.

In this chapter, we look at three different techniques found in distributed computing. Each technology has the potential of enhancing PropelXbi in its specific area. In each section, we explore what is the relation of this technology to PropelXbi as an implementation of XML pipeline processing system and how it could help in expansion of PropelXbi to a distributed computing environment.

Firstly, in section 7.1 we examine TupleSpaces, which stand as an alternative storage mechanism for intermittent documents between component transformations. TupleSpaces were designed to provide seamless distribution over multiple computers, which could be used for the distributed version of PropelXbi. Secondly, in section 7.2 we examine the JXTA Project, which is aimed at peer-to-peer computing, again potentially offering ways for the distribution of document processing. Finally in section 7.3 we look on the area of Grid computing. Grid technologies are meant to use multiple computers for common computational tasks, reflecting what is desired of distributed PropelXbi to do.

7.1 Tuple spaces

In this section, we look at TupleSpaces, which present an alternative storage space for intermittent documents between component transformations. In fact, it presents a complementary way of how to conduct distributed computing.

Firstly, we introduce the concept of TupleSpaces in section 7.1.1, then in section 7.1.2, we examine currently available implementations of TupleSpaces and at the end in section 7.1.3, we inspect how TupleSpace implementations relate to the current PropelXbi design and which implementation would be most suitable for integration in the PropelXbi architecture.

7.1.1 Concept of TupleSpace

Tuple Space were invented by David Gelernter in 1984 and was first described in “Linda in context” (Carriero & Gelernter 1989; Zhao 1998).

In essence, Tuple Space is a global shared memory (shared storage space) for lists of typed values called “tuples”. A simple model is used to access the tuple space, usually consisting of simple operations: write, take, read and optionally waitToTake, waitToRead, count and scan. Tuples themselves are accessed by pattern matching on their content (by associative addressing).

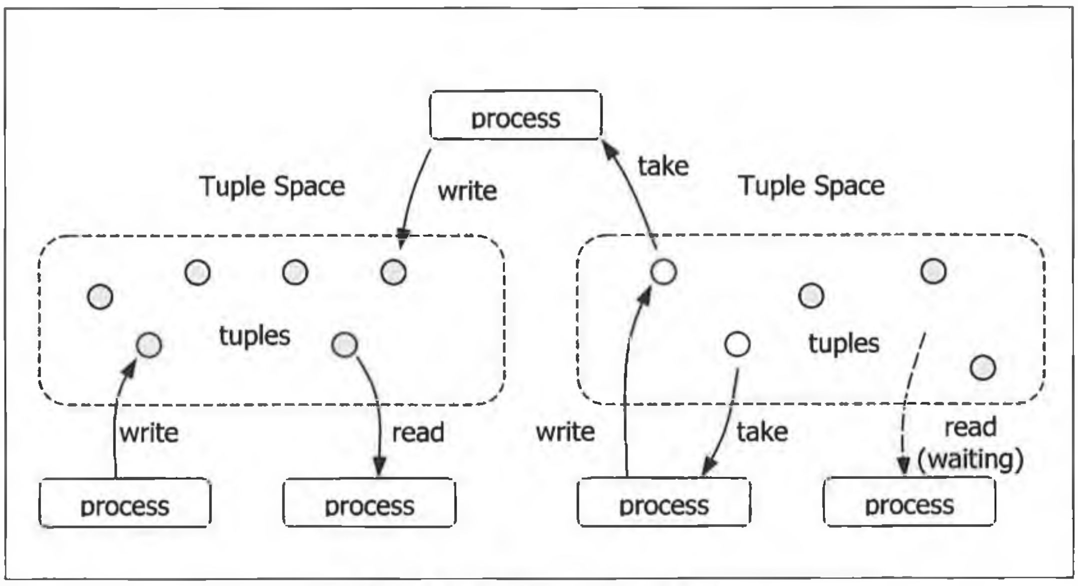


Fig. 7.1 Tuple space model

Access to tuples is by its nature asynchronous. When an application wants to read a tuple, it waits until the appropriate tuple is inserted into tuple space and then it is notified and the tuple is consumed. The whole system is implicitly event driven and allows concurrent access of multiple applications to the same tuple space. Another important feature of tuple spaces is that tuples are persistent, or can have set time of expiry and thus they can stay in tuple space long after the inserting application is gone.

Tuple space is meant to be distributed, hiding its distributed character from the user, to whom whole tuple space seems like one shared memory. A tuple space provides a Distributed Shared Memory (DSM) model, which gives the illusion of shared memory on top of a message passing system. Programming distributed applications, using distributed shared memory abstraction is less complicated than explicit message passing. The user thus, can just use a simple access model to a tuple space and leave all distributed data management (data localization, synchronization, persistence etc.) to the tuple space implementation.

In its present implementations, the tuple is represented by an object rather than a list of values, which was the original representation of the tuple by its inventors.

7.1.2 Implementations

At present, there are three major implementations of tuple spaces (CoverPages-TS; Strain). Firstly, Gelernter's original implementation in Linda language is presented and after that, a description of presently available implementations follows. Sun's JavaSpaces, GigaSpaces from GigaSpaces Technologies and IBM's TSpaces. Finally, we have a look at other implementations of tuple spaces.

Linda

The concept of tuple spaces was first implemented by the Linda language (or the Linda model in another view). The Linda language is a set consisting of a few simple operations, which embody the tuple space model of parallel programming. Linda was never a stand-alone programming language but was implemented as an extension to a base language (e.g. C, Fortran, C++), which yielded a parallel programming dialect, such as C-Linda.

The aim of Linda was to allow easy creation and coordination of multiple execution threads. This objective was achieved by providing a simple model for inter-process communication, independent of the programming language in which the processes are written.

In Linda, tuples were represented as lists of typed values and associative access to them was implemented by use of efficient hashing. Fundamental language primitives were:

`out`	-	Non-blocking write. Used to place tuple in tuple space.
`in`	-	Blocking read and delete. Used to remove a tuple from the tuple space. As it is a blocking statement, it waits until the matching tuple appears in the tuple space and is then executed.
`rd`	-	Blocking read.
`inp, rdp`	-	Non-blocking versions of `in` and `rd`
`eval`	-	Statement to create new process, being a 'live tuple', which after completion of its computing turns into an ordinary data tuple (this command was removed from today's tuplespace implementations)

Commands take a template as a parameter, which specifies the tuples on which the command should be executed. The template is just another tuple with assigned and un-assigned fields. The tuple matches if all the assigned fields in a template match identically and any un-assigned fields are matched by fields of the same type (in fact, un-assigned fields work as wildcards). If it happens, that there are more matching tuples for `rd` or `in`, one of the tuples is chosen non-deterministically.

Apart from `eval`, all commands mentioned above appear in today's implementations, usually with more intuitive names `write` and `read` for moving tuples to and from tuple space.

Linda was the first tuple space implementation and even at this early stage, it already had many of the advantageous features of the tuple space model.

As the tuple space model is machine and language independent, it inherently has a feature of portability on heterogeneous networks, scalability because of its simple architecture independent model and innate support for asynchronous communication, which is the most commonly used communication mechanism in parallel programming.

The fact that senders and receivers of tuples do not need to know anything about each other genuinely promotes an asynchronous uncoupled programming style. Another important feature of Linda, as well of other tuple space implementations, is data persistency, which allows for fail-over recovery if the system crashes.

The last and conceivably most important feature of Linda (and other tuple space implementations) is a simple API and a simple programming model which allows less and easier coding and intuitive understanding of how the system operates.

Nonetheless, Linda has its deficiencies. The main obstacle, hindering Linda from realistic use today is that it was created in 80's and 90's and today better implementations exist, which incorporated new findings discovered since Linda's creation. Even more, Linda's distribution isn't realistically available today.

One another problem of Linda comes from the environment for which it was primarily designed. It was predominantly designed as a parallel computing model for Local Area Networks (LANs) and because of this, it lacks a security model and any support for transactional execution.

JavaSpaces

JavaSpaces is Sun's implementation of tuple spaces (Zhao 1998; Shalom 2002b; Shalom 2002a; Sun Microsystems 2002d; Sun Microsystems). It is distributed as part of JINI Technology Starter Kit and its underlying communication mechanism is based on RMI, which constitutes the core of the JINI Network Technology. As indicated, JavaSpaces use Remote Method Invocation calls (RMI) as commands for placing tuples into a tuple space and retrieving them.

In contrast to Linda, in JavaSpaces a tuple is represneted by a Java Object, more concretely by an object implementing the `Entry` interface (`net.jini.core.entry.Entry`). `Entry` is Java's equivalent of Pascal's `record` or C's `struct`, thus being a collection of Serializable Java Objects.

Templates are implemented by Entry objects too. A matching tuple is an entry of the same type as a template with fields with assigned values matching exactly and with un-assigned fields used as wildcards. As an innovation, a match can also return a subtype of a template, which allows for polymorphism as entries like any other objects can have methods encapsulated in them. Entries are placed in a tuple space on a lease, meaning that the time of expiry can be set on them, after which they are removed from the tuple space.

Another new feature provided by JavaSpaces is support for distributed transactions keeping ACID properties. Operations can be issued either as singletons i.e. single individual operations or can be grouped into transactions where either all or none of them take place. Transformations in JavaSpaces can span multiple spaces meaning that various operations in one transaction can operate on various JavaSpaces. To achieve ACID properties, transformations are executed using a 2-phase commit technique.

As indicated, JavaSpaces implementation supports multiple spaces. Even though it implements multiple spaces, it doesn't natively support spaces spread over a cluster of computers.

To store entries, JavaSpaces use serialization (for this reason, the entry fields must be Serializable). This method though, may cause an overhead when serialization is executed every time an action is issued.

The usual operations in JavaSpaces are `write` – to write a copy of an entry to a tuple space, `read` – to get a copy of an entry from a tuple space and `take` to get a copy of an object and remove it from a tuple space. Non-blocking versions of `read` and `take` are `readIfExists` and `takeIfExists`. In addition to these standard operations, JavaSpaces provide two new commands – `notify` and `snapshot`.

`Notify` is used by a process to register as a listener to an event, which is fired when an entry matching with a specified template is inserted into the JavaSpace. `Snapshot` is used for performance optimisations reducing the overhead of repetitive entry serializations.

JINI JavaSpaces Service Specification states "Persistence is not a required property of JavaSpaces technology implementations" (Sun Microsystems 2002d). Nevertheless, Sun's reference implementation provided in JINI Technology Starter Kit claims it does provide persistency.

The reliance of JavaSpaces on RMI as a communication mechanism may be a source of its drawback. Execution completely based on RMI may be slow and repetitive serializations may cause non-negligible overheads. Another imperfection of JavaSpaces is that it lacks any security model and does not genuinely support spreading JavaSpaces over a cluster of computers.

GigaSpaces

GigaSpaces is commercial implementation of Sun's JavaSpaces specification developed by GigaSpaces Technologies (GigaSpaces Technologies 2002c; GigaSpaces Technologies 2002b; GigaSpaces Technologies 2002a; Shalom 2002a). In contrast to Sun's reference implementation discussed in previous section, GigaSpaces contains various important enhancements.

They are summarised in the following list, each discussed in more detail successively

- Space clustering
- External database JDBC support
- Web services support
- Batch operations
- Administration and configuration GUI and command-line interface

Space clustering provides the possibility to create multiple GigaSpaces possibly on different physical machines and to access them through one unified point of access called clustered proxy. Clustering allows for three main features – Replication, Fail-over and Load-balancing.

Replication means that data is partially or fully replicated (mirrored) on multiple spaces so that clients accessing space don't have to connect to one specific machine but can access the one that is closest to them. Replication also makes possible data recovery in case of a breakdown of some of the replicated spaces.

Fail-over is a security technique allowing redirection of transaction execution to a different space when the original target space crashes or is unavailable for any other reason, like for example maintenance.

Finally, load-balancing conducted by cluster proxy implements policy-based work distribution to member spaces so that computing resources are used as efficiently as possible.

As GigaSpaces aren't always used in clustered environment, it also provides supports for embedded space and local transactions. Embedded space is GigaSpace created in the same JVM as the application (client) and because of this proximity, the client can use local (not remote) operations, reducing overhead associated with distributed remote transactions execution.

JDBC support is part of the technology used in GigaSpaces to implement persistency. In GigaSpaces, space can be either transient, when all data is kept in memory or persistent when data is stored in a database. Persistent spaces can use either an internal database or any external database which is JDBC compliant like for example Oracle, DB2, MS SQL etc.

Support for web services means, that internal space of GigaSpaces can be accessed from web using web protocols e.g. UDDI, WSDL and SOAP.

GigaSpaces extends the standard JavaSpaces API in three major ways.
Firstly batch operations were added, so that performance can be improved by executing operations on groups of entries in one step. Added methods are:

`writeMultiple`	-	Writes a group of entries in one access to space
`readMultiple`	-	Returns a group of entries that match a specified template
`takeMultiple`	-	Takes a group of entries that match a specified template

Secondly, administration API was added to allow management and control of spaces. For example it allows, creating spaces on the fly, destroying them, checking their content etc. This API is used to provide an administration and control GUI and a command-line interface.

The last enhancement is a semantic extension allowing execution of updates on entries already placed in a space and to define actions, which should be performed at defined points of executions. These actions are called "filters" and their placings are `On_Init`, `Before_Write`, `After_Write`, `Before_Read` and `After_Read`.

As two last technical enhancements, GigaSpaces supports database entry indexing which speeds up searching for matching entries and support for queues, where entries are added to the tail and are taken from its head. GigaSpaces can contain any number of named queues, which can be created and shared on the fly.

TSpaces

TSpaces is an implementation of tuple spaces developed by IBM research team at Almaden Research Center (IBM; IBM; Wyckoff et al. 1998; Zhao 1998; Lehman, McLaughry & Wyckoff 1999; Lehman et al. 2001; IBM). It is not an implementation of JavaSpaces specification (like GigaSpaces) but has a lot in common.

Like GigaSpaces, it allows tuplespaces to be either transient, stored in memory or persistent, stored in an internal DB2 database. It doesn't provide means for connecting to any other external database though.

Again, like GigaSpaces, it provides extended API allowing handling groups of tuples:

`multiWrite`	-	Equivalent to `writeMultiple`
`multiUpdate`	-	Updates all matching tuples
`Scan`	-	Equivalent to `readMultiple`
`consumingScan`	-	Equivalent to `takeMultiple`

In addition, it offers other commands, from which the most interesting are:

`Update`	-	Updates matching tuple, being already in tuplespace
`countN`	-	Returns number of matching tuples

Concerning transactions, TSpaces does provide transactions support, but with the limitation that transactions cannot span over more TSpaces servers. Concerning distribution of TSpaces over more machines, it is promised to implement it in Enterprise TSpaces. Enterprise TSpaces aim to provide replication over more servers for fault-tolerance and scalability. At present, this version is not available, and because of IBM's decision not to continue in support of TSpace project anymore, it is questionable when Enterprise TSpaces will be released (Lehman). In the current version, TSpace server can be located only on one machine.

Compared to GigaSpaces, TSpaces offer several enhancements. In brief they are:

- Extended querying facility
- Security based on Access Control Lists
- Possibility to dynamically define new commands
- Notification on `update` event

These features are discussed in the following paragraphs in greater detail.

TSpaces leverage features of the underlying DB2 database. Tuples are implemented as vectors of fields and every named field stored in the database is indexed for faster access. Indexing also allows for range queries, like for example "get all tuples with first field 'record' and second containing value in range <1-100>". The next feature under the extended querying heading is the ability to construct more complicated queries by joining templates, using the logical connections AND and OR.

The last improvement of the query mechanism is focused on special kind of tuples. Tuples can contain at most one XMLField, which stores an XML document. This document can then be queried using a subset of the XQL language (XML query language). The returned result is then a tree of tuples mirroring a DOM representation of an XML document.

Even though TSpaces allow saving XML documents as parts of tuples, XML documents are not meant to be the sole data saved in tuplespace. Authors say that XML support was added as a repository for Web Services Descriptions thus allowing web services discovery.

As indicated in paragraph above, a tuple in TSpaces is implemented as `Tuple` object, which is a vector of `Field` objects. A `Field` specifies type, value and name of a field. Apart from exact match, matching on templates can also return any subtype of a template, as a `Tuple` is an object itself. This object-oriented extension of the original concept of tuple matching can be found in all current Tuplespace implementations.

The second worthy enhancement provided by TSpaces is a basic security technique based on Access Control Lists (ACLs). ACL sets permissions for various groups, into which users can be assigned. A user can be assigned to any number of groups and authorities of these groups are then assigned to him. Every time a client calls a command, it submits client username and password which are used by the server to determine who is requesting execution of the command and to which group he belongs and consequently what are his authorities.

In addition TSpaces provides the possibility of defining new commands, which use already existing functions, and load them dynamically to the server.

TSpaces also extends the notification mechanism, so that an application can register for an `update` event (not only for `read` event as in GigaSpaces)

Apart from discussed major improvements, there are other rather minor improvements.

TSpaces provides a property which can be set when a new tuplespace is created, which defines whether or not, a tuplespace keeps FIFO ordering. Meaning, that when a client matches on a group of tuples, it receives the one which was put into the space first. By default, tuple selection is unordered and this extension was made for cases, when tuple order plays important role.

An interesting added feature is the possibility to reference a file by a URL and to write just this reference to a tuple. This is useful, when the file to be transported is big and writing it to tuplespace would use a lot of space and would take a long time. With URL referencing, the receiver retrieves file from the sender only when it actually reads the tuple and consumes it.

In contrast to JavaSpaces implementations, TSpaces doesn't use RMI, but implement its own remote procedure call (RPC) mechanism. TSpaces' RPC uses serialization and Java TCP/IP sockets, which implementation may be faster than general purpose RMI, as it is designed specifically to be used for TSpaces.

The last of TSpace improvements is support for local operations, which is implemented in GigaSpaces too. Local operations are used when the client and the TSpaces server run in the same JVM, where the use of direct calls significantly reduces execution time compared to the time taken by RPC calls.

The main limitation of TSpaces appears to be its lack of support for clustering and therefore does not provide replication and load-balancing capabilities.

Other Tuplespace implementations

There are other projects aimed at implementing the tuple spaces, but none of them are in the utilizable form or they are focused on different areas of computer world than our research. As a reference we mention two of them, namely jxtaSpaces and Ruple (Collab.Net; Quovadx).

JxtaSpaces is a project aimed at implementing a Distributed Shared Memory (DSM) service on a JXTA peer-to-peer platform by implementing tuple spaces. The JxtaSpaces Project is still in the design stage and there isn't any implementation available yet. Furthermore, the project proposal doesn't mention any intention to implement security and data persistency.

Ruple is a project of Rogue Wave Software, which came with an interesting idea. In contrast to other tuplespace implementations, it chose the XML document architecture as the mechanism for tuple implementation. It was meant to be an "Internet based space" for which XML is an ideal technology. As an Internet based space, it was accessible by HTTP and SOAP protocols, backed by security model based on X.509 digital certificates. Unfortunately, this promising project was discontinued without any articulated reason.

In following section, we discuss relation of TupleSpaces to XPipe's implementation PropelXbi.

7.1.3 Relation of TupleSpaces to PropelXbi

In this section, we examine how tuple spaces and their implementations relate to the PropelXbi architecture and how it could be incorporated into it.

To see how the TupleSpaces relates to PropelXbi, let's recall the current PropelXbi architecture. From the high-level view, there are two fundamental parts of PropelXbi.

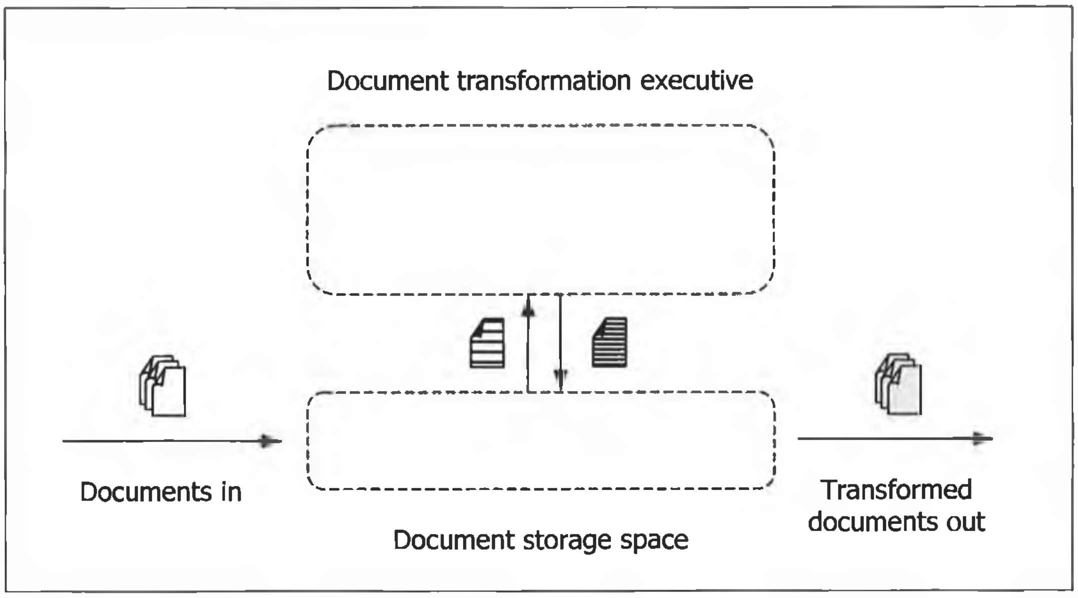


Fig. 7.2 High level PropelXbi architecture

The first part is a document storage space, which acts as a storage space for documents in their intermittent stages. The second part is a document transformation executive, which executes the actual transformation of documents.

In PropelXbi, the document storage space is implemented by a JMS queue and a document transformation executive by a pool of MDB objects.

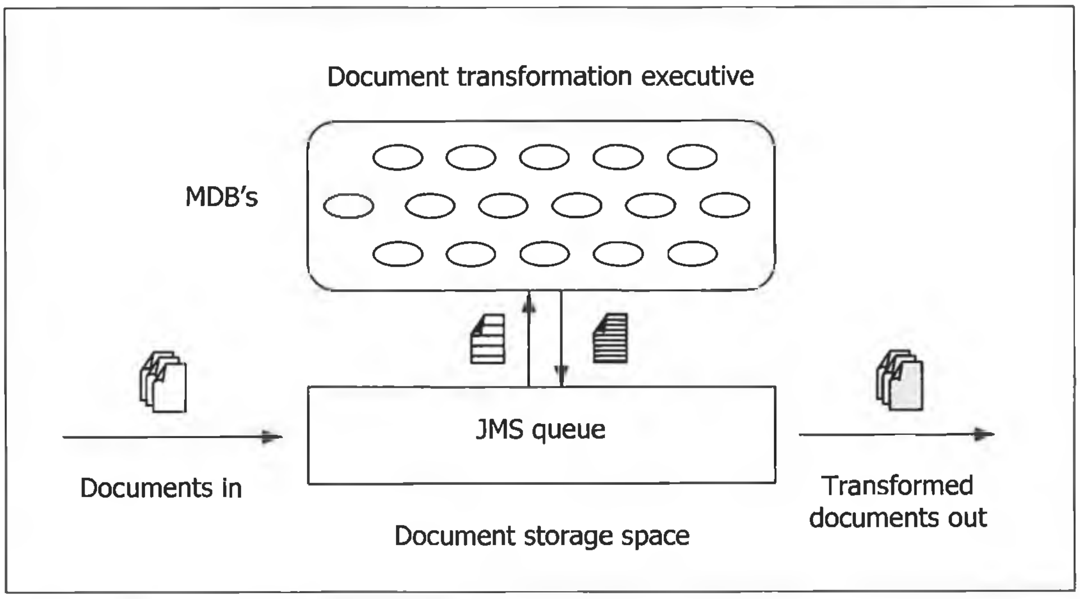


Fig. 7.3 Current PropelXbi architecture

The JMS queue was chosen to implement the document storage space, because it possesses three important architectural features. Firstly, it allows asynchronous communication, secondly, it is an event driven architecture and thirdly it provides data persistence for cases when a system crashes. However, persistence of JMS queues is limited to static queues only (created on start-up), not allowing dynamic creation of persistent queues on the fly.

Tuple space is in its nature a distributed shared storage space and thus it is an alternative to JMS queues as a document storage space implementation in PropelXbi.

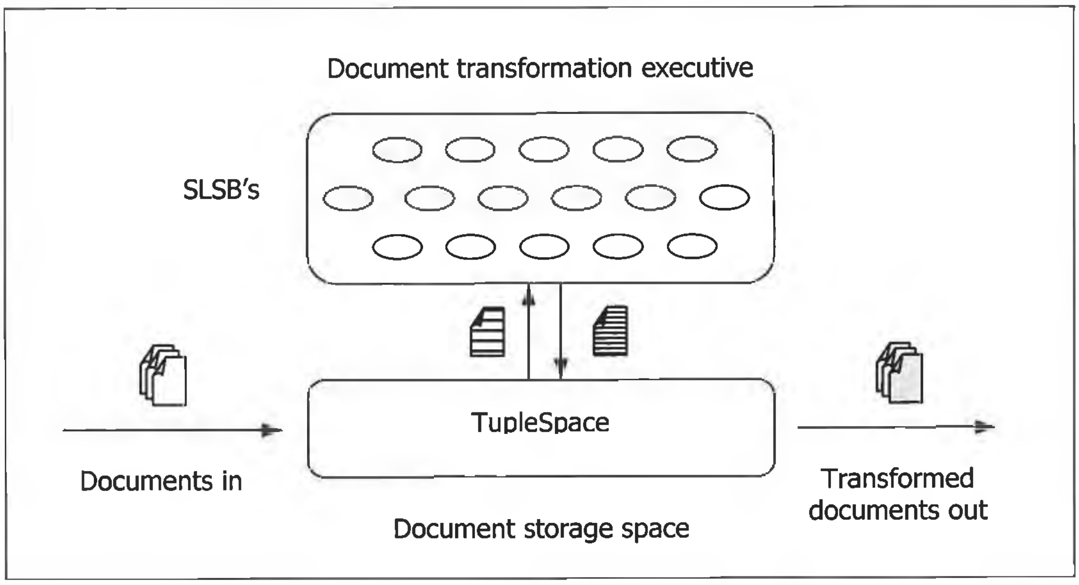


Fig. 7.4 PropelXbi architecture with TupleSpaces

Message driven beans are activated by notifications fired from JMS queue when appropriate message arrives. If we replace JMS queues with a TupleSpace, the bean activation mechanism would need to be changed, as there will be no messages coming from the JMS queue. MDB's would need to be replaced by Stateless Session Beans (SLSB's) performing `read` command on the TupleSpace. As `read` is a blocking command, the beans would wait until the message (tuple) appears in the TupleSpace and would then be activated. This substitution would preserve the event-driven nature of beans activation and behaviour.

TupleSpace possesses all three architectural features of JMS queues – an asynchronous communication mechanism, an event driven architecture and data persistence capabilities. In addition, it allows space distribution over number of computers, thus allowing data replication for fail-over recovery facility, scalability and locality based load balancing. However, this clustering facility is available only in GigaSpaces.

Feature qualities of JMS queues and TupleSpces are comprehensibly summarised in following table.

	JMS Queue	TupleSpace
Asynchronous	YES	YES
Event driven	YES	YES
Persistent	YES *	YES
Distributed	YES	YES
Clustered	no ***	YES **

Tab. 7.1 JMS Queue and TupleSpace comparison table

Note:

* Persistence of JMS queues is limited to static queues only. GigaSpaces and TSpaces provide persistence of also dynamically created queues.

** All tuple space implementations support multiple tuplespaces being placed on different machines. However, only GigaSpaces provide space clustering i.e. unified access to spaces spread across number of computers with transparent view of one big tuplespace.

*** Some JMS implementations provide queue replication facility for clusters of computers. However, this feature is not required by JMS Specification (it's not even mentioned) and can't be counted on.

Apart from the storage space distribution facility, replacing the JMS queue by tuplespace implementation would provide other enhancements:

- Individual pipelines can be implemented by individual separate tuplespaces.

  This would largely simplify monitoring of pipelines.

  Moreover, by allocating one tuplespace per pipeline, the whole PropelXbi architecture would become more close to original XPipe architecture, which is a move in good direction, as discussed in section 4.2.

- Architecture with tuplespace would open the possibility to implement PropelXbi@Home

  PropelXbi@Home is the concept, that computers in company would process documents in their idle time, corresponding to the concept of

SETI@Home, where home computers process radio data in time when they aren't used

- Pipeline separation would allow the option to set priorities on different pipelines.

  This concept is discussed in greater detail in paragraphs below.

The rationale to introduce this feature is to handle the situation where a company uses many pipelines of which some do higher priporty work than others.

For example, there might be one pipeline, which executes transformation of records of large legacy database. This job needs to be done, but without having strictly limited timespan. In contrast, there might be another pipeline for processing documents, which are of high importance to company and timely completion of their transformation is the company's main interest. In this scenario, it would be convenient to have an option to set different priorities of individual pipes.

If each pipeline was represented by one tuple space, storage space separation would lead to different ways for SLSB's to retrieve documents to process. In a single space scenario, they would access a single tuple space from which they would get documents. With separate spaces for every pipeline, SLSB's would have to poll individual spaces one by one.

As transformation beans are meant to have as little handling logic as possible, so that their sole function is transformation execution, retrieving documents from individual pipelines can be implemented by other handling beans whose only function would be providing documents to transforming workers.

New architecture then would look like the following figure, which is described underneath.

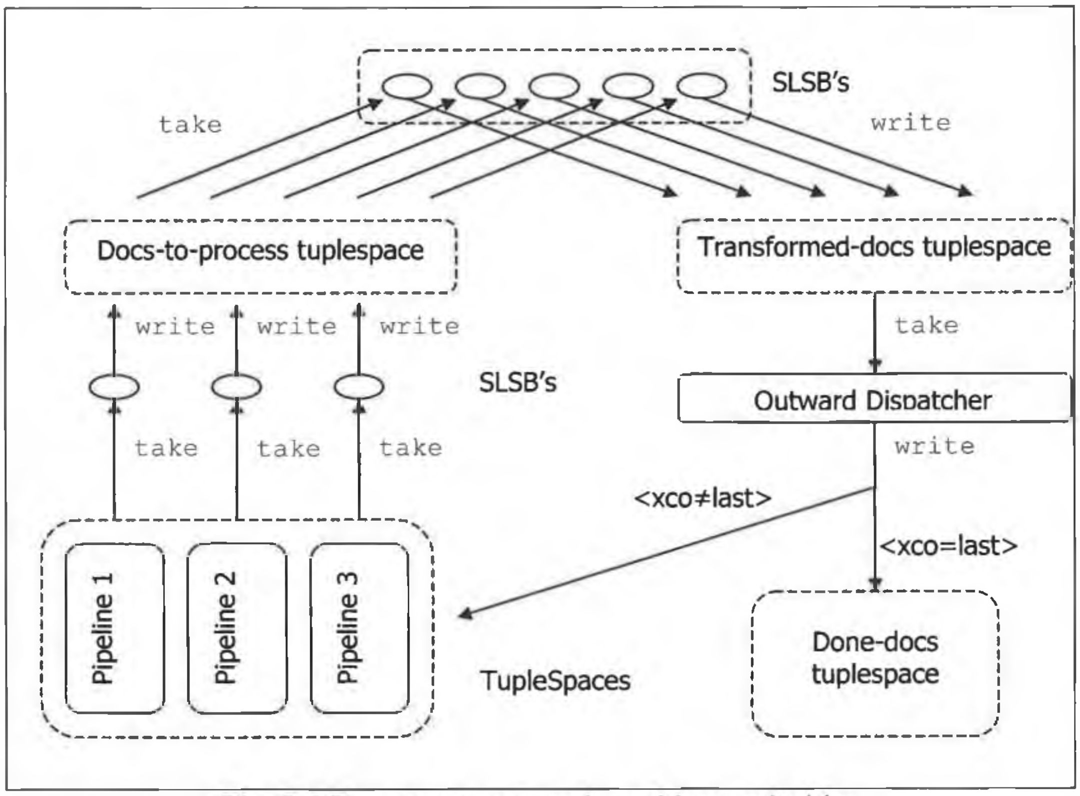


Fig. 7.5 Documents processing without priorities

This new architecture looks fairly complex, but in its core it only consists of three simple steps. Firstly, it retrieves documents from pipeline spaces, then it transforms the documents and finally writes them back to tuple spaces.

At the start of the document transformation cycle, SLSB's are assigned to every pipeline's tuplespace and wait for any document to appear in the tuplespace. When it arrives, the SLSB takes it out and places it into the docs-to-process tuplespace. This space works as a storage space for documents ready to be processed by transforming beans.

Transforming SLSB's watch the docs-to-process space and when new a document arrives it is immediately consumed, if there are any free SLSB's. The SLSB then performs the document transformation and outputs the document into the transformed-docs space.

The transformed-docs space is observed by a Outward Dispatcher, whose role is to place partially transformed documents into the tuplespace representing an appropriate pipeline and take out documents which are completely transformed and place them in a done-docs tuplespace. The done-docs tuplespace is watched by out-routing beans, which take care of shipping out the processed documents. As `take` and `write` are the only blocking operations used, the whole system stays event driven as in the architecture with the JMS queue.

When we decide to add priority handling, the whole system gets a bit more complicated. Additional logic needs to be added between the pipeline spaces and the transforming beans to apply priority selection. The Inward Dispatcher object, realising this policy would select the documents according to priority settings and pass them to a new docs-to-transform tuple space, from which the SLSB's get the documents to process.

The architecture with the priorities module plugged in is shown on figure below.

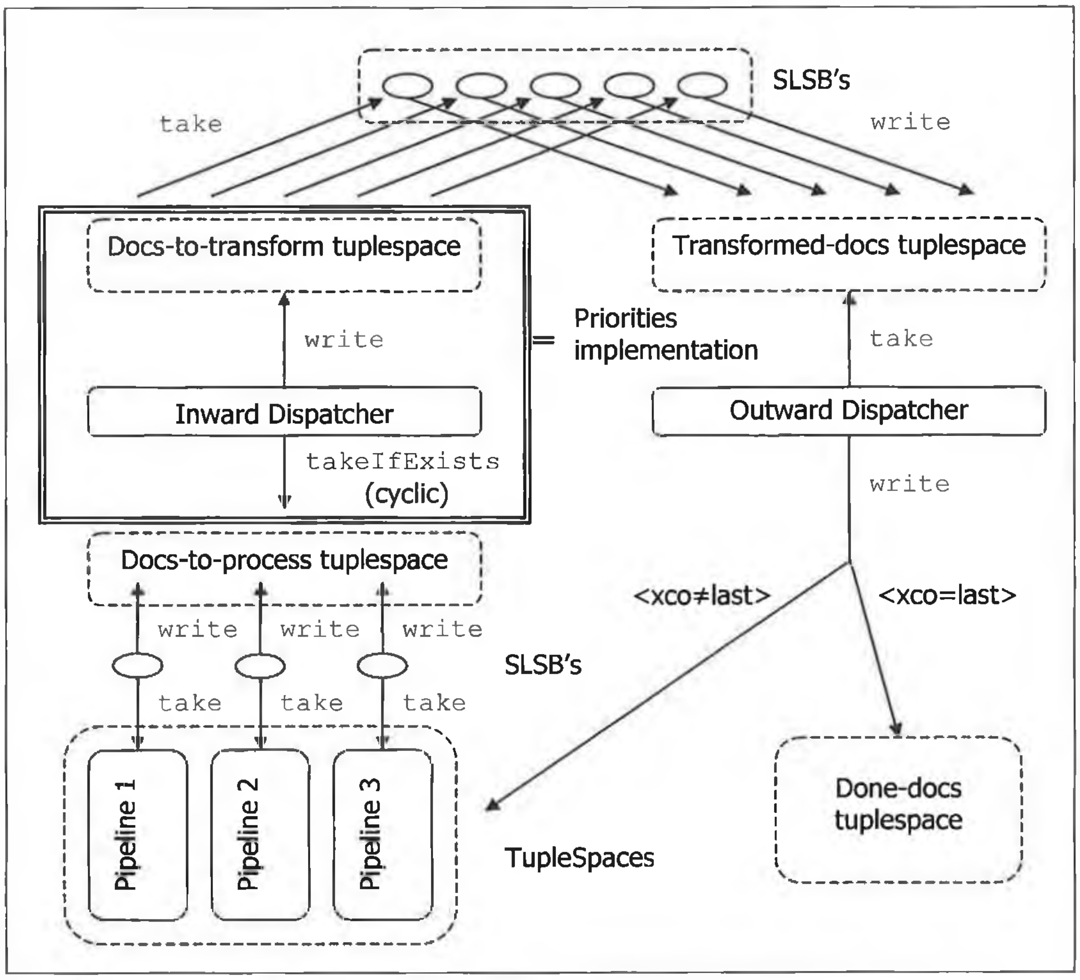


Fig. 7.6 Documents processing with priorities

Priorities could be implemented by setting the default priority of pipelines to the highest value. If a user decides not to use priorities, then all pipelines would have the same priority and would be treated equally.

If priorities are used the Inward Dispatcher would check documents of individual pipelines in the docs-to-process space, and if the priority of some pipeline was lowered, it would simply skip checking its documents accordingly to the level of its priority.

To give an example, the default priority (highest) may be set to 10. If there were two pipelines in system, one with a default priority of 10 and another with priority of 7, in 10 checking rounds, documents of the first pipeline would be checked 10 times and documents of the second pipeline 7 times (the dispatcher would skip 3 checks of the second pipeline documents).

Information about pipeline priorities can be saved in another dedicated tuplespace containing configuration information of all tuple spaces in the system.

Before we look at individual TupleSpace implementations we can envisage, how the operating code of the SLSB's would look if TupleSpace was used instead of the JMS queue. The code below is conceptual code for three types of beans, which exist in PropelXbi – in-routing bean, worker and out-routing bean and further code for additional beans, which would have to be created.

```
In-router:  in_space.take(msg)
            prepare(msg)
            pipeline.write(msg)

Worker:     docs-to-transform.take(msg)
            transform(msg)
            transformed-docs.write(msg)

Out-router: done-docs.take(msg)
            ship-out(msg)
```

additional handling beans:

```
Pipe listener:       pipeline.take(msg)
                     docs-to-process.write(msg)

Inward dispatcher:   apply priorities and
                     select msg
                     docs-to-transform.write(msg)

Outward dispatcher: transformed-docs.take(msg)
                     if (xco=last)
                        done-docs.write(msg)
                     else
                        pipeline.write(msg)
```

Tab. 7.2 Conceptual code of architecture with TupleSpaces

This code overview shows that operation code stays simple and no extended querying facilities are needed.

If we ultimately decide to employ TupleSpaces in PropelXbi, we would have to select the most suitable TupleSpace implementation. For this purpose, the following table is provided. It summarises the features of examined TupleSpace implementations and juxtapose JMS queue system currently used in PropelXbi. Linda can't be reasonably considered as possible candidate, but is included for reference.

	JMS	Linda	JavaSpaces	GigaSpaces	TSpaces
Language	Java SE	Various	Java SE	Java SE	Java SE
Tuple implem.	JMS Message	Vector of values	Serializable Object	Serializable Object	Serializable Object

Method invocation	RMI + local calls	Function call	RMI call	RMI + local calls	RPC[1] + local calls
Speed	RMI + local calls	?	RMI	RMI + local calls	RPC[1] + local calls
Asynchro-nous	Y	Y	Y	Y	Y
Event driven	Y	Y	Y	Y	Y
Persistent	Y	-	Y[2]	Y	Y
Distribu-ted[3]	Y	-	Y	Y	Y
Clustered spaces	-	-	-	Y	-
Security	-	-	-	-	ACL
Transaction support	Y	-	Y[2]	Y	Y
Leasing[4]	Y	-	Y	Y	Y
Message ordering	Y	-	-	Y[5]	Y
Free	Y	Y	Y	NO	NO

Tab. 7.3 Features of TupleSpace implementations

Notes:

[1] TSpaces uses in-house implementation of RPC using Serialization and Java TCP/IP Sockets.

[2] Feature isn't required by specification

[3] "Distributed" meaning that individual tuplespaces can be placed on different computers and can be used at the same time

[4] "Leasing" means setting limited time of life on tuples. After lease expires, the tuple is removed from tuplespace.

[5] In addition to message ordering, GigaSpaces provide support for queues.

Java SE – standard edition of Java. Not using J2EE architecture.

For JMS: Space = JMS queues, Tuple = JMS message, Method invocation = adding and retrieving messages to/from queue

Each implementation also has some unique features not found in other implementations. These advanced features are listed in following table.

Advanced features	Linda	JavaSpaces	GigaSpaces	TSpaces
Command eval (creates tuple as a process)	Y	-	-	-
Command notify	-	Y	-	-
Command snapshot	-	Y	-	-
Local transactions	-	-	Y	Y
Tuple indexing	-	-	Y	Y
Batch transactions	-	-	Y	Y
Clustering	-	-	Y	-
JDBC	-	-	Y	-
Web services support	-	-	Y	Y[1]
Extended querying	-	-	-	Y
ACL security	-	-	-	Y
Dynamic definitions of new commands	-	-	-	Y

Tab. 7.4 Advanced features of TupleSpace implementations

Note:

[1] Support for Web services is provided by additional TSpaces services suite package, which creates a service layer on top of TSpaces architecture.

From the three possible candidates (JavaSpaces, GigaSpaces, TSpaces), the selection can be narrowed to GigaSpaces and TSpaces as they provide database persistency and local transactions support, which are not provided by Sun's reference implementation.

Both selected implementations provide API for batch operations, ACID transactions support, support for web services, aforementioned support for local calls and both implement tuple indexing for improved speed of access to tuples.

In areas where they differ, TSpaces provide Access Control List based security, extended querying possibilities and faster execution mechanism (by using in house developed RPC technique). GigaSapces, on the other hand, enable space clustering and persistence binding to external database.

When comparing these two feature sets, clustering facilities appear more important than features provided by TSpace. Extended querying is by all means a helpful tool, but for the needs of PropelXbi, the standard template matching mechanism is sufficient.

For these reasons, GigaSpaces looks like the most suitable TupleSpace implementation for integration into PropelXbi as distributed document storage space.

7.2 Project JXTA

In this section, we will look on Project JXTA as a platform for distributed peer-to-peer computing. First we introduce the JXTA project and its primary goal in section 7.2.1. Secondly, we present the high-level design of Project JXTA in 7.2.2 and after that, in section 7.2.3 we look at the actual architecture which implements Project JXTA's objecive. Finally, in section 7.2.4, we look at the interrelation of Project JXTA and PropelXbi.

7.2.1 Project JXTA Introduction

Project JXTA is an open-source project, originally initiated as Sun's internal research project, opened to public in April 2001 (Sun Microsystems; Verbeke et al.; Gong 2001a; Gong 2001b; Sun Microsystems 2001d; Sun Microsystems 2001c; Sun Microsystems 2002e; Traversat et al. 2002; Collab.Net; Collab.Net; Sun Microsystems 2003b). The primary goal of Project JXTA is to provide a platform with the basic functions necessary for a peer-to-peer (P2P) network.

Project JXTA defines a set of protocols for ad hoc peer-to-peer computing allowing peers implementing these protocols to communicate and collaborate with any other devices on the network implementing JXTA protocols.

As JXTA aims at standardization of peer-to-peer messaging system it defines only the protocols, not their implementations.

To let you better understand what Project JXTA is, we can use an analogy with Open GL. Similar to Project JXTA, Open GL is a specification which provides a common platform for applications written in different languages on different hardware and software configurations. Whereas JXTA is common networking platform, Open GL provides a common programming platform for computer graphics.

Both Project JXTA and Open GL themselves are just platform specifications and the actual services / functions are implemented in different languages. As these implementations are based on common specifications they all provide the same methods with the same behaviour, hiding actual implementation on any given HW and SW configuration from the user.

7.2.2 Design of Project JXTA

In order to achieve the goal of providing a universal peer-to-peer communication platform, Project JXTA set its three key objectives.

- Interoperability – any P2P system built on JXTA can talk to each other
- Platform independence – JXTA technology is independent of programming languages, network protocols, hardware and software platforms
- Ubiquity – JXTA can be deployed on any device with a digital heartbeat

The protocols defined by Project JXTA implement JXTA's core concept of establishing a virtual network on top of existing physical networks, hiding their underlying physical topology.

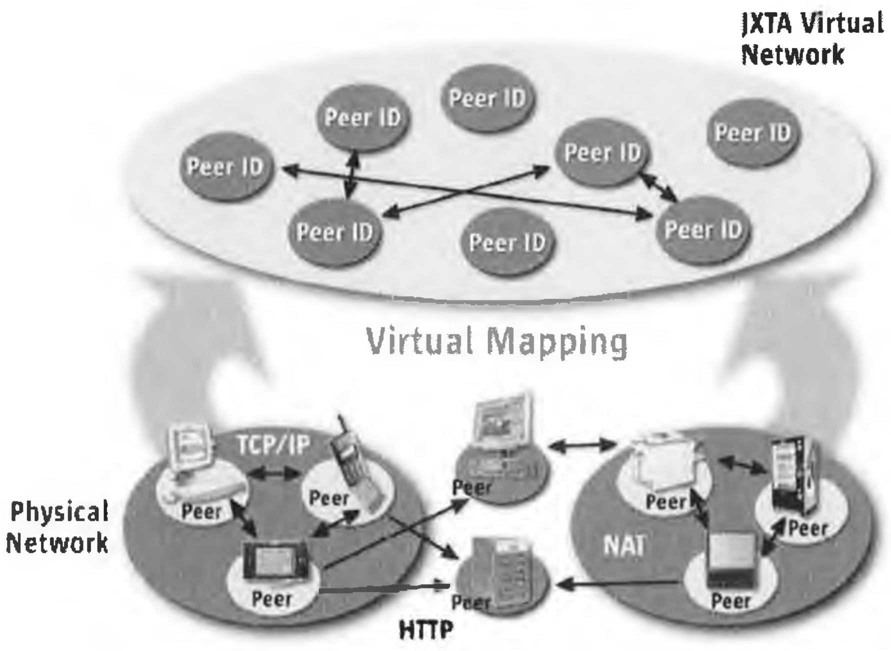


Fig. 7.7 JXTA virtual network

Because the JXTA network is virtual, any peer can interact with other peers and other resources directly, even when some of the peers and resources are behind firewalls and NATs (Network Address Translation – network security technique) or are on different network transports (Collab.Net).

In a nutshell, protocols defined by Project JXTA standardise the manner in which peers

- Discover each other
- Self-organize into peer groups
- Advertise and discover peer services
- Communicate with each other
- Monitor each other

Because these protocols are independent of both programming language and transport protocols, heterogeneous devices with completely different software stacks can interoperate with one another.

7.2.3 JXTA architecture

The architecture of Project JXTA is rather complex and therefore we divide its description into three sections proceeding from the top level view down to the more technical details. First we introduce the architectural layers of Project JXTA, then we examine the components which constitute the JXTA architecture and finally we look at the protocols on which the architecture is built.

Note: Information, diagrams, descriptions and definitions in this section (7.2.3) were taken from Project JXTA: Java Programmer's Guide (Sun Microsystems 2001d) as they succinctly and clearly explain the JXTA technology architecture.

1) JXTA Architectural layers

The Project JXTA software architecture can be divided into three layers, as shown on following figure:

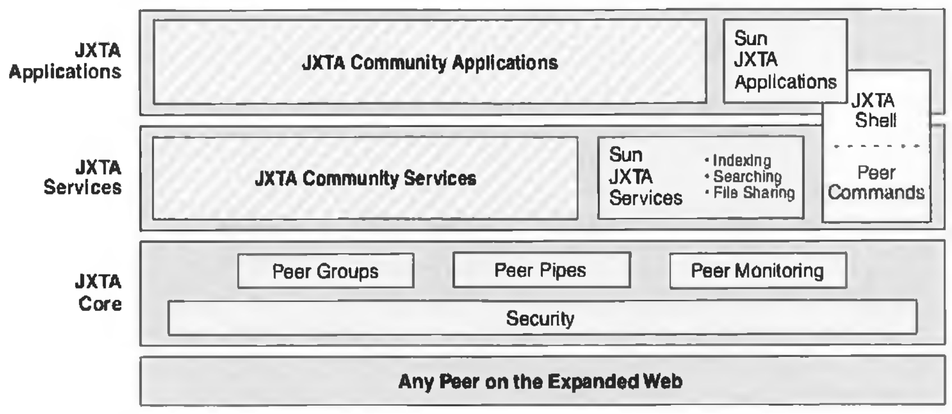


Fig. 7.8 Project JXTA architectural layers

These layers are:

Platform Layer (JXTA Core)

The platform layer, also known as the JXTA core, encapsulates minimal and essential primitives that are common to P2P networking. It includes building blocks to enable key mechanisms for P2P applications, including discovery, transport (including firewall handling), the creation of peers and peer groups, and associated security primitives.

Services Layer

The services layer includes network services that may not be absolutely necessary for a P2P network to operate, but are common or desirable in the P2P environment. Examples of network services include searching and indexing, directory, storage systems, file sharing, distributed file systems, resource aggregation and renting, protocol translation, authentication, and PKI (Public Key Infrastructure) services.

Applications Layer

The applications layer includes implementation of integrated applications, such as P2P instant messaging, document and resource sharing, entertainment content management and delivery, P2P Email systems, distributed auction systems, and many others.

The boundary between services and applications is not rigid. An application to one customer can be viewed as a service to another customer. The entire system is designed to be modular, allowing developers to pick and choose a collection of services and applications that suits their needs.

2) JXTA Components

In a nutshell, the JXTA network consists of a series of interconnected nodes, or *peers*. Peers can self-organize into *peer groups*, which provide a common set of *services*. Examples of services that could be provided by a peer group include document sharing or chat applications. JXTA peers advertise their services in XML documents called *advertisements*. Advertisements enable other peers on the network to learn how to connect to, and interact with, a peer's services. JXTA peers use *pipes* to send *messages* to one another. Pipes are an asynchronous and unidirectional message transfer mechanism used for service communication. Messages are simple XML documents whose envelope contains routing, digest, and credential information. Pipes are bound to specific *endpoints*, such as a TCP port and associated IP address. These concepts are described in greater detail in the following sections.

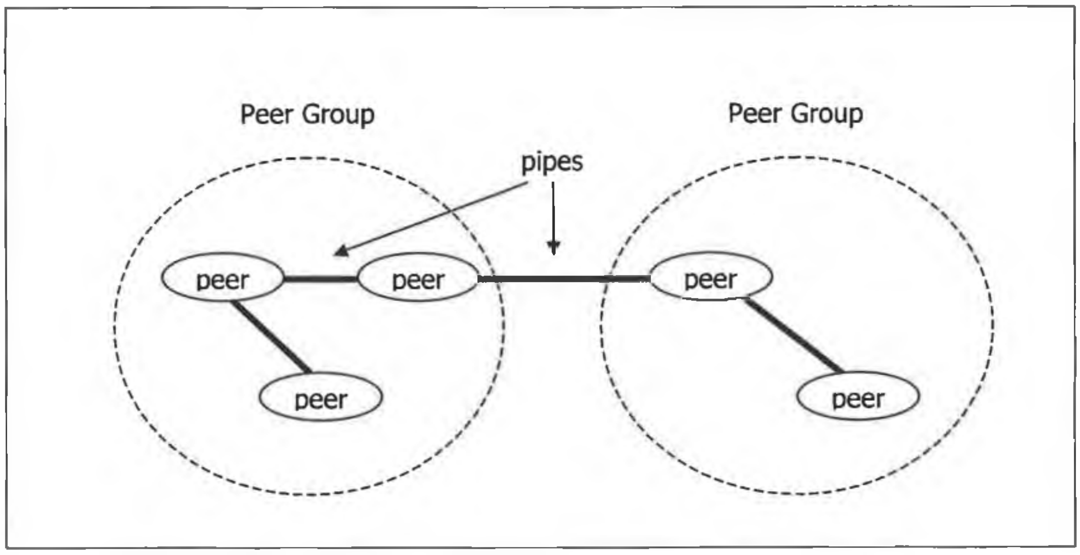


Fig. 7.9 Network of JXTA peers

Peers

A peer is any networked device that implements one or more of the JXTA protocols. Peers can include sensors, phones, and PDAs, as well as PCs, servers, and supercomputers. Each peer operates independently and asynchronously from all other peers, and is uniquely identified by a Peer ID.

Peers publish one or more network interfaces for use with the JXTA protocols. Each published interface is advertised as a peer endpoint, which uniquely identifies the network interface. Peer endpoints are used by peers to establish direct point-to-point connections between two peers.

Peers are not required to have direct point-to-point network connections between themselves. Intermediary peers may be used to route messages to peers that are separated due to physical network connections or network configuration (e.g., NATs, firewalls, proxies). Peers spontaneously discover each other on the network to form transient or persistent relationships called peer groups.

Peer groups

A peer group is a collection of peers that have agreed upon a common set of services. Peers self-organize into peer groups, each identified by a unique peer group ID. Each peer group can establish its own membership policy from open (anybody can join) to highly secure and protected (sufficient credentials are required to join).

Peers may belong to more than one peer group simultaneously. By default, the first group that is instantiated is the Net Peer Group. All peers belong to the Net Peer Group. Peers may elect to join additional peer groups. The JXTA protocols describe how peers may publish, discover, join, and monitor peer groups; they do not dictate when or why peer groups are created.

Creation of peer groups allows to create secure environment, (peer groups can implement their own security policy), scoping environment (groups can establish a local domain of specialization, like for example working on one specific task) and monitoring environment (peers of a group can monitor each other).

Network services

Peers cooperate and communicate to publish, discover, and invoke network services. Peers can publish multiple services. Peers discover network services via the Peer Discovery Protocol.

The JXTA protocols recognize two levels of network services – Peer Services and Peer Group Services. A Peer Service is accessible only on the peer that is publishing that service. If that peer should fail, the service also fails. Multiple instances of the service can be run on different peers, but each instance publishes its own advertisement.

A Peer Group Service is composed of a collection of instances (potentially cooperating with each other) of the service running on multiple members of the peer group. If any one peer fails, the collective peer group service is not affected (assuming the service is still available from another peer member). Peer group services are published as part of the peer group advertisement.

JXTA defines a core set of peer services. Core services are Discovery Service for searching for other peers, services and groups, Membership Service for joining peer groups, Access Service for securing access to services, Pipe Service for managing pipe connections between peers, Monitoring Service for peer monitoring and Resolver Service enabling sending generic query requests to other peers.

Not all core services must be implemented by every peer group. A peer group is free to implement only the services it finds useful, and rely on the default net peer group to provide generic implementations of non-critical core services.

Messages

A message is an XML document that is sent between JXTA peers; it is the basic unit of data exchange between peers. It is an ordered sequence of named and typed contents called message elements. Thus a message is essentially a set of name/value pairs. The content can be an arbitrary type.

The use of XML messages to define protocols allows many different kinds of peers to participate in a protocol. Because the data is tagged, each peer is free to implement the protocol in a manner best-suited to its abilities and role. If a peer only needs some subset of the message, the XML data tags enable that peer to identify the parts of the message that are of interest. For example, a peer that is highly constrained and has insufficient capacity to process some or most of a message can use data tags to extract the parts that it can process, and can ignore the remainder.

Pipes

JXTA peers use pipes to send messages to one another. Pipes are an asynchronous and unidirectional message transfer mechanism used for service communication. Pipes are indiscriminate; they support the transfer of any object, including binary code, data strings, and Java technology-based objects.

Pipes are virtual communication channels and may connect peers that do not have a direct physical link. In this case, one or more intermediary peer endpoints are used to relay messages between the two pipe endpoints.

The pipe endpoints are referred to as the input pipe (the receiving end) and the output pipe (the sending end). Pipe endpoints are dynamically bound to peer endpoints at runtime. Peer endpoints correspond to available peer network interfaces (e.g., a TCP port and associated IP address) that can be used to send and receive message. JXTA pipes can have endpoints that are connected to different peers at different times, or may not be connected at all.

Pipes offer two modes of communication, point-to-point and propagate. A point-to-point pipe connects exactly two pipe endpoints together: an input pipe on one peer receives messages sent from the output pipe of another peer. A propagate pipe connects one output pipe to multiple input pipes. Messages flow from the output pipe (the propagation source) into the input pipes. All propagation is done within the scope of a peer group. That is, the output pipe and all input pipes must belong to the same peer group. The JXTA core also provides secure unicast pipes, a secure variant of the point-to-point pipe.

Additional types of pipe services can be built using the basic core pipes. For example, the current J2SE platform binding (implementation) includes bi-directional pipes.

Advertisements

All JXTA network resources - such as peers, peer groups, pipes, and services - are represented by an advertisement. Advertisements are language-neutral metadata structures represented as XML documents. The JXTA protocols use advertisements to describe and publish the existence of peer resources. Peers discover resources by searching for their corresponding advertisements, and may cache any discovered advertisements locally.

Each advertisement is published with a lifetime that specifies the availability of its associated resource. Lifetimes enable the deletion of obsolete resources without requiring any centralized control. An advertisement can be republished (before the original advertisement expires) to extend the lifetime of a resource.

3) JXTA Protocols

JXTA defines a series of XML message formats, or protocols, for communication between peers. Peers use these protocols to discover each other, advertise and discover network resources, for inter-peer communication and messages routing.

All JXTA protocols are asynchronous, and are based on a query/response model. A JXTA peer uses one of the protocols to send a query to one or more peers in its peer group and it receives zero, one, or more responses to its query depending on how many (if any) other peers can send a reply.

JXTA peers are not required to implement all core protocols; they only need implement the protocols they will use. The current Project JXTA J2SE platform binding supports all six core JXTA protocols. The Java programming language API is used to access operations supported by these protocols, such as discovering peers or joining a peer group.

7.2.4 Relation of JXTA technology to PropelXbi

Project JXTA itself is a set of protocols. To actually use it, some implementation (called binding) of JXTA has to be chosen. As PropelXbi is written in Java, we decided to use a Java implementation called Project JXTA 2.0 J2SE platform binding.

JXTA 2.0 J2SE binding provides an API implementing the Project JXTA specification. It allows a user to write peer to peer applications using JXTA high-level concepts (peers, peer groups, pipes, services and advertisements), hiding actual low-level implementation from the user (e.g. it hides communication protocols and actual format of JXTA messages).

In addition to the standard services defined in the Project JXTA specification, Java binding offers two additional enhancements :

- bi-directional pipes
- secure pipes

Note: Pipes are still assumed to be un-reliable (as stated in JXTA specification). Their actual implementation may use the special characteristics of the network protocol they run on, but it's not required by the Project JXTA specification. Java binding uses TCP/IP which is in most cases reasonably reliable, but still can not be considered fully reliable.

JXTA's place in PropelXbi architecture

As mentioned earlier, JXTA provides a network communication system. Therefore, in the PropelXbi architecture, it would act as a communication mechanism between the document storage space (JMS queue) and the document transformation executive (Enterprise JavaBeans – EJB's).

The current PropelXbi architecture consists of a JMS queue realising a document storage space and pools of MDB's realising a document transformation executive. Communication between these two primary components is implemented by JMS messages and a messaging infrastructure provided by the JMS system.

The following figure depicts the present PropelXbi implementation showing MDB pools as clients and the JMS queue as the server. This is a reasonable view as it is the setting in which the distributed PropelXbi would run in a real-world scenario (the only change is that the server could potentially host MDB pool too)

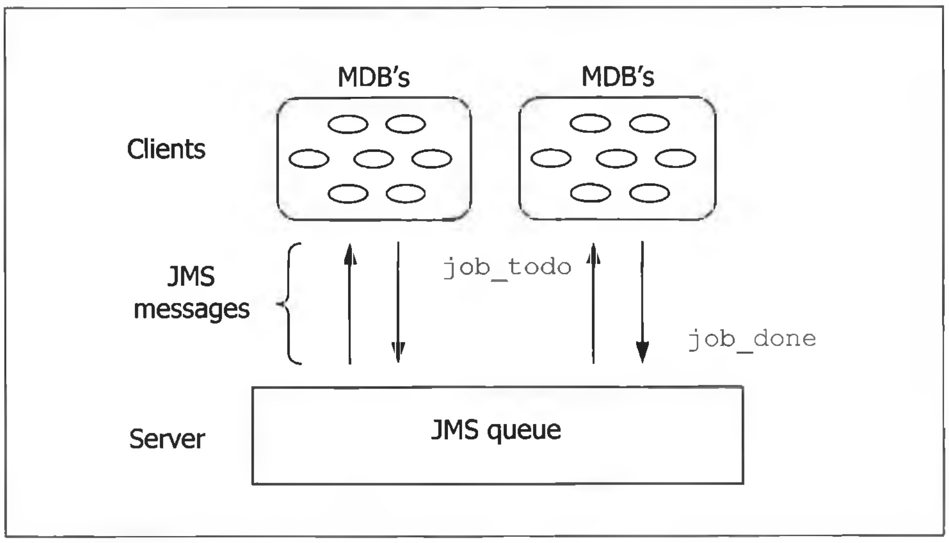


Fig. 7.10 Current PropelXbi architecture with JMS communication system

If JXTA was integrated into PropelXbi architecture, it would replace the JMS messaging system providing communication between the document storage space and the EJB pools by using bi-directional JXTA pipes.

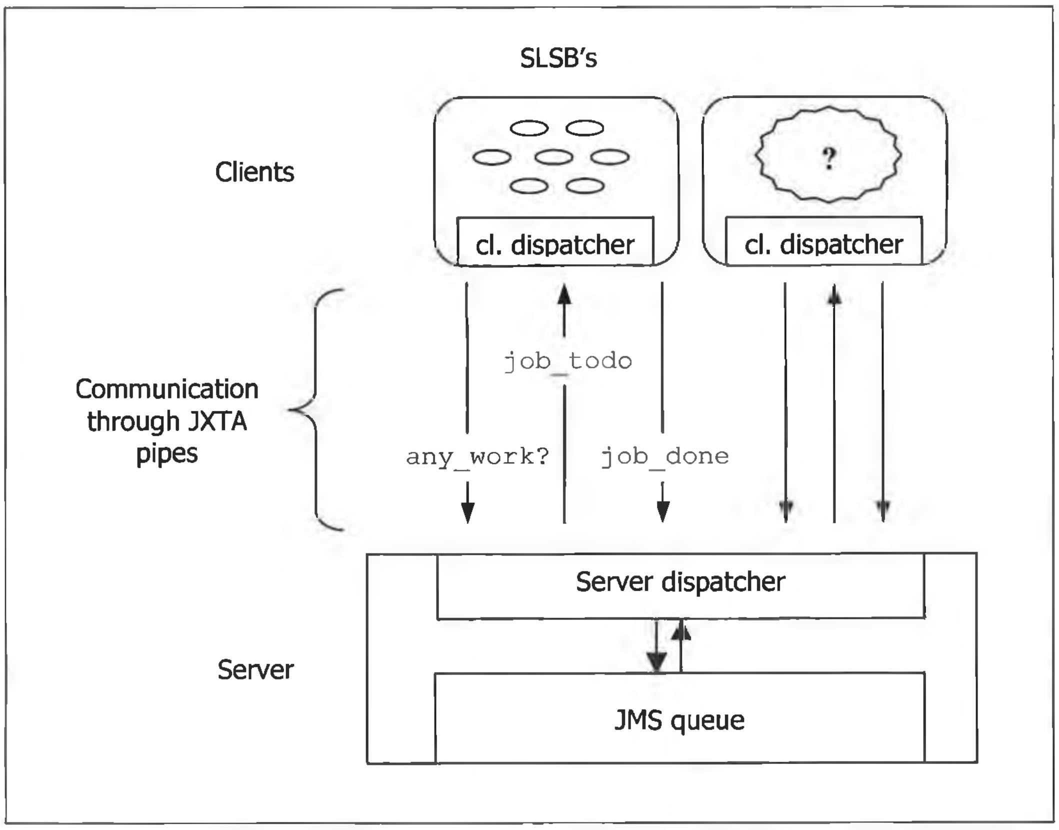


Fig. 7.11 PropelXbi architecture with JXTA communication system

JXTA pipes allow transportation of any data structure containing any desired message representation.

MDB's can only work when supplied with JMS messages and thus they would need to be replaced by Stateless Session Beans (SLSB's), which are another, less specialised type of Enterprise JavaBean.

As JXTA messaging is not bound to any particular language, clients can be written in any language for which there is JXTA implementation and can use any transforming technology they desire, as long as they understand what to do with incoming document.

At the moment there are JXTA implementations in Java SE, Java ME and C. Other language implementations (Perl, Python, Smalltalk and Ruby) are open projects in different stages of progress.

By removing JMS messaging we loose inherent work distribution infrastructure (as EJB's can not directly access the JMS queue anymore). Because of that we would need to build some other way of document passing from server to clients. This is implemented by Client and Server Dispatchers.

The client dispatcher polls the Server dispatcher to see if there are any documents to process. If they are, they are sent to the dispatcher which passes the work to SLSB's. When the SLSB finishes processing of the document, it passes the document back to the client dispatcher, which sends it back to the server dispatcher. The server dispatcher retrieves the document from the JXTA pipe and inserts the transformed documents back to the queue.

Comparison of current and JXTA-augmented architecture

When we look on what needs to be done to replace JMS messaging by JXTA we can immediately see the main drawback. We must take out innate messaging system and then build it again with JXTA technology, abandoning JMS messaging infrastructure which is there already available.

Incorporating the JXTA communication technology in PropelXbi is un-natural as PropelXbi already contains an innate messaging system, which is used in the present PropelXbi architecture as it was intended.

Further advantages and disadvantages of JXTA incorporation into PropelXbi are listed in following summary and discussed in greater detail afterwards.

Advantages

- Clients can be written in any language (for which there is JXTA implementation)
- Client's transformation technology is not bound to JMS and EJB
- Document server could be dynamically discovered by JXTA lookup

Disadvantages

- Not reliable
- Not persistent
- Not part of J2EE package (not natively designed to be used with other J2EE technologies)
- Redundant dispatching logic

The advantages brought by employing JXTA come mainly from its language and platform independence. As JXTA is a language-independent communication system, participating clients can be written in any language, for which there is JXTA implementation. However, at the moment there are only Java and C bindings.

For the same reason of technology-independence, clients are not bound to EJB and JMS paradigm, and their document transforming technology can be any other (like for example monolithic code, proprietary pipeline systems …). However, there is one problem that would arise if we wanted to use multi-language clients. The problem is how to convey information about what should be done with the passed document. At the moment, passed messages contain the document to process and the number of the pipeline stage. The stage number identifies the component (piece of code) which carries out actual transformation. With different languages used by the receiving clients, there would have to be multiple sets of components in different languages, so that every client would be able to execute them.

An alternative method of conveying information about the document transformation process, is to invent some formal language, which would describe the transformation. This would be a blind alley though, as formal languages are never flexible enough (definitely less flexible than actual code) which would result in limiting the range of transformations which can be carried out on the document.

Another advantage of using JXTA technology is that it would be possible to advertise server pipe in the JXTA network and any client would be able to look it up using the JXTA discovery mechanism. This would allow clients to join the data transformation process at an arbitrary time. However, the implementation of PropelXbi@Home using JXTA as communication mechanism is hindered by JXTA's serious faults which are discussed in following paragraph.

The faults of JXTA come mainly from the authors' aim at having as general communication mechanism as possible. In order to have a general communication system, they chose the lowest common denominator in the area of communication technologies, resulting in the mere requirement, that pipes have to be unidirectional and unreliable. Unreliability is serious problem, as data commonly gets lost when transferred by JXTA communication pipes. Even though, JXTA 2.0 J2SE binding provides bi-directional and secure pipes, reliable pipes weren't implemented yet. As PropelXbi needs to have reliable messaging, this fault rules out use of JXTA in its architecture.

Another JXTA's drawback is that it doesn't provide data persistency and so if system crashes, all the data that was in transit between the sender and the receiver is lost.

Project JXTA is an open source project and as it is, it was not included in J2EE platform. This gives advantages to the other alternative technologies (like JMS for example), which are part of J2EE, as they are designed to natively cooperate with other components of J2EE package.

The last indicated flaw was already mentioned earlier on the beginning of the section. By incorporating JXTA into PropelXbi, we unnecessarily build work dispatching logic, which is already available as a native part of JMS technology.

All these mentioned flaws overweight JXTA's advantage of relative language independency and makes it unsuitable for incorporating into PropelXbi architecture.

7.3 Grid computing

In this section we look at the area of Grid computing. At first, in 7.3.1 we introduce the concept of the Grid, present definition of the Grid and describe different types of Grids that exist today. In the following section 7.3.2 we present generic Grid architecture and in the final section 7.3.3 we examine how PropelXbi can be enhanced using Grid computing technologies. Section 7.3.3 is further subdivided into sections describing generic architecture of distributed PropelXbi, section where we examine existing Grid applications available today. In the closing section, we discuss most suitable candidates for deployment of PropelXbi into a distributed computing environment.

7.3.1 Concept of Grid

The concept of Grid computing is to enable communities to use and share geographically distributed resources as they pursue common goals. These resources represent computational resources, storage devices, special-purpose devices and any other computing devices, which may be useful for any community user (Foster & Kesselman 1999; Foster & Kesselman 2001).

A Grid itself is defined as "an ensemble of geographically-dispersed resources interconnected by fast network that appear to the end-user as a single seamless computing and communication environment." (Weissman 2002)

The grid computing environment has many unique characteristics distinguishing it from other more conventional computing environments. The essential trait of Grid is that its constituting computing devices are physically distributed often over very large areas without any central point of control and without knowledge of global state of the whole Grid. Other distinguishing feature is high heterogeneity of the computing devices, as computing is carried out on various hardware and software platforms. The fact that Grids are often created in collaboration of several institutions exposes another unique Grid feature, which is that Grid resources are owned by multiple different entities with different usage policies. Finally, one of the most important Grid features is, that resource availability can change with time, as resources are added and removed. (Foster & Kesselman 1999)

There are three major reasons why organisations decide to build Grid networks. Pursued goals are increased computational performance, access to widely distributed data and establishment of new enhanced multi-institutional services. These three distinct goals lead to three different types of Grids, whose architecture reflects different organisational goals (Krauter, Buyya & Maheswaran 2002).

1. Computational Grid – the goal is improved computational performance by using the current idle and remote computational resources. Computational Grid provides higher aggregate computational capacity than can be provided by any single machine.

2. Data Grid – the goal is to get access and share widely distributed data (across companies, states, continents …). Data Grids are aimed to allow synthesises of new information from data repositories distributed over large area networks (e.g. massive data mining) In contrast with computational Grids, data Grids provide special infrastructure for data access and storage management.

3. Service Grid – the goal is to create enhanced services that can not be provided by any single machine. Example of such services are collaborative computing – allowing dispersed teams to interact and work together, real-time multimedia applications and on-demand computing – using grid capabilities to meet peak short-term requirements for resources that can not be cost-effectively or conveniently located locally.

There are other different classifications of Grid types (Foster & Kesselman 1999; Sun Microsystems 2002f; Jacob 2003), but in the core, all classes they define fall in one of the classes defined above.

The Computational Grid architecture is chosen for deployment of PropelXbi on the Grid, to increase computational power and thus to increase the number of processed documents per time unit.

The following section discusses the general Grid architecture common to all three types of Grid.

7.3.2 Grid architecture

General Grid architecture was described in "The Anatomy of the Grid" (Foster, Kesselman & Tuecke 2001). It consists of layered model similar to layered Internet protocol architecture.

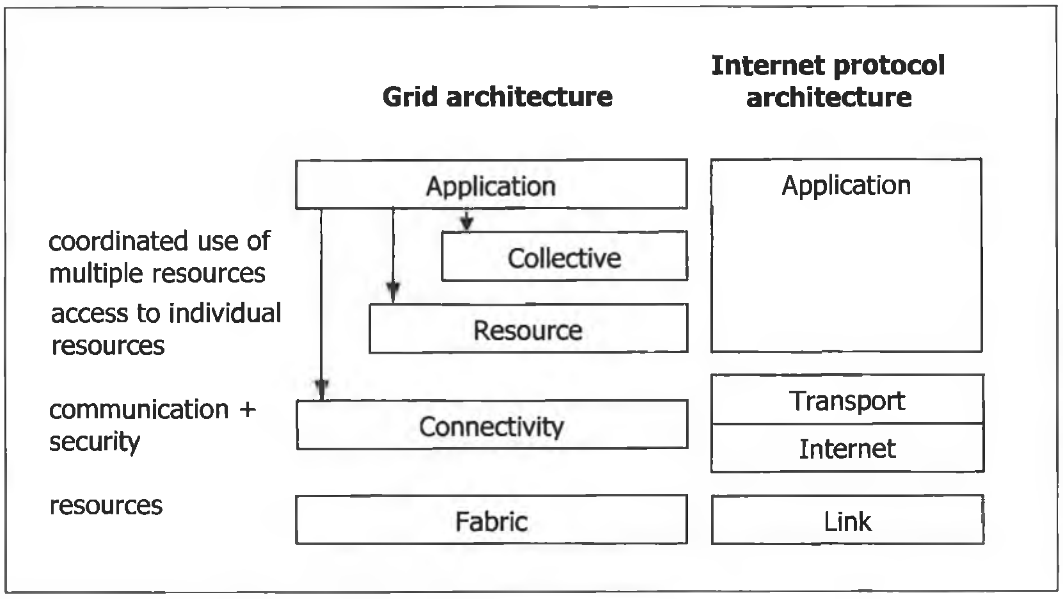


Fig. 7.12 Grid layered architecture

The components within each layer share common characteristics and can build on capabilities and behaviours provided by any lower layer.

The Fabric layer provides interfaces to local resources such as computational resources, storage systems, networks and sensors. Requests through unified fabric layer interface are mediated to fabric components, which implement local resource-specific operations on involved resources.

The Connectivity layer handles network communication and security. It defines core communication and security protocols required for Grid-specific network transactions. Apart from transport and security, the Connectivity layer also handles routing and naming.

The Resource layer builds on the communication and authentication protocols of the Connectivity layer to define protocols for secure negotiation, initiation, monitoring, control, accounting, and payment of sharing operations on individual resources. Resource layer protocols are concerned entirely with individual resources and hence ignore issues of global state and atomic actions across distributed collections, such issues are the concern of the Collective layer.

The Collective layer handles coordinating of use of multiple resources. While the Resource layer is focused on interactions with a single resource, the Collective layer defines services and protocols, which are not associated with any specific resource but rather are global in nature and capture interactions across collections of resources. Services commonly provided by the Collective layer applications are co-reservation and co-allocation, workflow management, replication, global monitoring and metainformation directories.

The final layer is an Application layer consisting of Grid applications. Grid applications can use any services defined at any underlying layer, accessing the ones that best suits their needs.

7.3.3 PropelXbi on Grid

In this section we look on how PropelXbi can be enhanced using Grid computing technologies.

Praxis showed, that the use of Grid is beneficial when the problem to be solved exhibits any of following features – data parallelism, task parallelism and data-flow (Foster & Kesselman 1999). XPipe paradigm and PropelXbi have all these features – data parallelism in mutual independence of documents being transformed, task parallelism in independence of individual pipeline stages (which were intentionally designed as black boxes that can be used as stand-alone entities) and data-flow in the mere concept of pipelines and data flowing from one component to another. All these features make PropelXbi an ideal candidate for the employment of Grid technologies.

First, in "Generic PropelXbi Grid architecture" we present generic high-level view of how PropelXbi can be deployed in distributed environment using Grid technologies. Afterwards in "Current Grid technologies survey" we review the Grid applications which are currently available and finally in "Most suitable candidates for PropelXbi" we present the most suitable candidate applications which can be used in PropelXbi in its expansion to distributed computing world.

Generic PropelXbi Grid architecture

The reasons people decide to use Grid architecture are either to increase computing power or throughput, access widely distributed data or improve fault tolerance of system. The first and last goals – increased throughput and fault tolerance can be achieved using Grid-based approach described later in the section. But first, let's have a look on what is current PropelXbi architecture.

As said many times before, in essence, PropelXbi consists of message queue, storing documents and pool of Message Driven Beans (MDB's) transforming them. As the message queue and MDB pool are two separate entities, they can be placed on different computers. Naturally, this separation leads to a straightforward extension of PropelXbi for distributed computing by placing multiple MDB pools listening to one message queue on different machines.

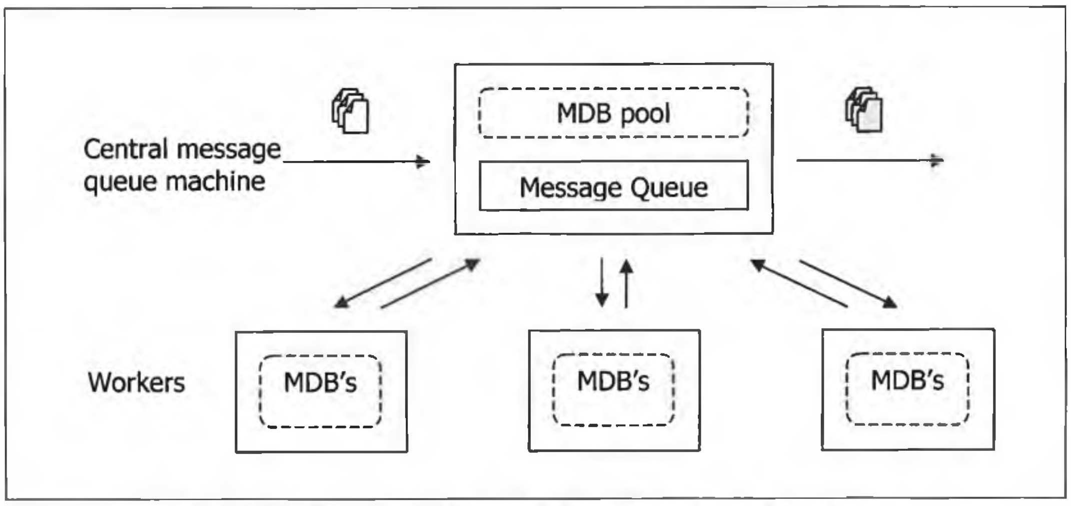


Fig. 7.13 Distributed PropelXbi non-Grid architecture

Machines with MDB pools then serve as "workers" providing execution power to main queue machine, which may contain MDB pool as well. MDB's in pools listen to a remote central queue and the transportation of messages is executed via a standard RMI serialization protocol.

The power of this system can be simply increased by adding another pool of MDB's. The process of taking a document form queue, transforming it and returning back to the queue is a single transaction. Therefore, the sudden removal of an MDB pool doesn't cause the system to crash. When an MDB pool is removed, all transactions involved are stopped and rolled back and all involved documents re-submitted to other pools. This feature ensures a basic level of fault tolerance.

Even though, this solution is straightforward and requires very little change to current PropelXbi system, it has numerous drawbacks.

The first problem is that this solution has very bad scalability. As there is only one central message queue, we can add only a limited number of MDB pools, after which the load on the central queue would be too high and the queue would be unable to effectively deal with such large number of simultaneous listeners.

Moreover, the singularity of a central message queue can create a performance bottleneck. If there was large number of documents to process and the queue was already congested handling previously arrived documents, there may be free idle MDB's which stay unutilised because the central queue would be too busy and incapable of distributing the new work.

The next drawback lies in mechanism used for document transformation, which is RMI. RMI is a general remote transport mechanism and because of its general focus it exhibits long transportation times. Time spent in transporting documents to and from the main queue would be unacceptably long in comparison to time spent in actual document transformation.

The last problem, which relates to the previous one, is the inefficient document handling. One document needs to be transferred between the queue and the MDB at least twice as many times as its number of pipeline stages. Again, time spent by the document transfer creates a big overhead in comparison with the time spent by the actual document transformation.

The problems mentioned above can be avoided if we use a Grid-based architecture as depicted on following figure.

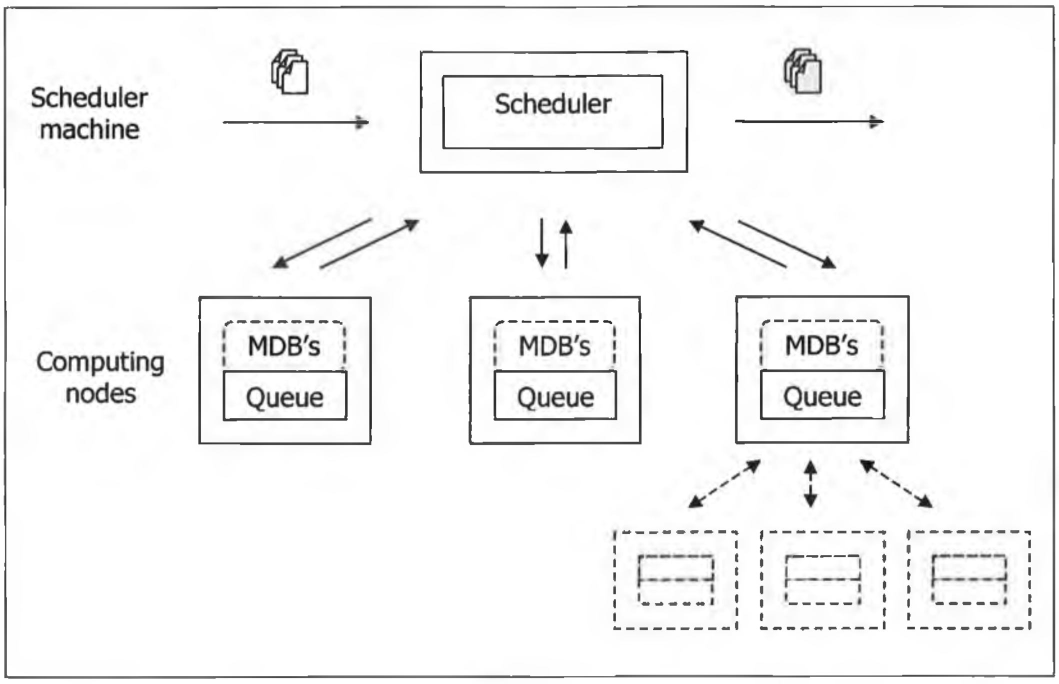


Fig. 7.14 Distributed PropelXbi Grid-based architecture

At the top of the diagram is a Scheduler, which receives documents and distributes them to computing nodes according to their workload and optionally other additional policies. Computing nodes are normal PropelXbi queue & pool machines. If it is found that it is beneficial to use non-Grid architecture as a computing node, it can be added in, as the scheduler is oblivious as to how documents are processed in the computing nodes. Because there aren't restrictions on how documents should be handled on computing nodes, hierarchical architectures of computing nodes can be built.

The scheduler, in fact, performs a role of document distributor/collector being the document-level analogy of the Scatter/Gather, which works in the scope of document segments. The distribution policy of scheduler can be adjusted to meet different needs, so for example it is possible to specify the preferred computing platform or that some computers should be used only if their workload is less than 5% and were idle for at least 15 minutes. By this, it is possible to implement PropelX@Home (being parallel to SETI@Home project) using computing cycles of idle workstations to carry out document transformations.

As the scheduler keeps track of execution progress, it provides automatic fault-tolerance facility by re-submitting jobs to new machines if the original machines became unavailable (either because of crash or any other reason).

Additionally, Grid-based architecture solves the problem of communication overhead encountered in previously discussed non-Grid installation. Documents are sent to computing nodes only once and received back only when they are transformed, so the number of needed transfers per document decreases to two. As documents are submitted directly to computing nodes, they are processed in local environment and MDB's can communicate with queue through local calls without the need of lengthy remote invocations.

Current Grid technologies survey

After presenting the generic view of PropelXbi's deployment on a Grid, we will examine what actual applications currently exist in Grid computing world. As Grid computing originated in academic circles, there are many toolkits, libraries, programs and applications in different stages of development covering different layers of Grid architecture. As different architectural layers pose different programming challenges on developers, programs often specialize on one layer, leveraging services of applications from layers underlying them.

Our survey is divided correspondingly into two parts. In first part (page 131), we examine resource management tools covering Fabric, Connectivity and Resource layer. Surveyed toolkits are Legion and Globus. In second part (page 137), we have a look on schedulers (or resource brokers), which handle job management and implement the Collective layer of Grid architecture. Some of the schedulers prefer not to depend on resource management services of other software and rather use their own means of resource management, which incorporates all the architectural layers from Fabric to Collective.

As different schedulers may have different performance goals, the survey of them is accordingly further subdivided into a section of High-throughput schedulers (page 138) and a section of High-performance schedulers (page 152). More detail about this division is given in section "Schedulers" introducing schedulers survey.

Resource management

The area of Grid resource management is governed by two major toolkits – Legion and Globus. Their main responsibilities are resource discovery and keeping track of resource properties, state and availability.

Legion

Legion (Foster & Kesselman 1999; Legion; Natrajan, Humphrey & Grimshaw 2001; Avaki Corporation; Avaki Corporation 2003b) is an object-based system providing abstraction of a grid as a single powerful virtual machine. By its nature it is a middleware layer between operating system and other Legion resources. While providing single machine abstraction, it transparently takes care of scheduling of applications on available processors, managing data migration, data cashing and data transfers. Moreover it performs fault detection and fault management and ensures that user's data and physical resources are adequately protected.

Legion is designed to work on a variety of architectures (Intel, IBM, HP ...) and to run applications on multiple platforms (Windows, Unix, Linux …). Legion supports legacy applications without requiring any change to source or object code. Applications do not have to be "Legion-aware", i.e., they need not access Legion objects. For Legion-aware applications, Legion provides a C++, C, Java and Fortran interface.

Legion Architecture

As mentioned above, Legion is reflective object based system consisting of classes and metaclasses (which are classes whose instances are classes itself). "Reflective" here means that system is able to retrieve information about its objects at run time.

Architecture of Legion is built upon four essential concepts:

Firstly, "Everything is an object". Every entity which is part of computing system is represented by a Legion object. Objects represent both software (applications, users ...) and hardware resources (processors, storage spaces ...). Objects in Legion system are mutually independent and communicate with each other via non-blocking method calls.

Secondly, "Classes manage their instances". The duty of object management is assigned to the class objects themselves, by which Legion implements an architecture in which central management hub is not needed. Class objects have system-level responsibilities, meaning that they cater for new instance creation, scheduling of instance execution, activation and deactivation and providing information about their current location, when other clients want to communicate with them.

Thirdly, "Users can provide their own classes". Legion allows users to define and build their own class objects and by that Legion programmers can determine and even change the system-level mechanisms that support their objects. Legion's reference implementation provides default implementations of class objects and of all the core system objects. Users can use them, but aren't required to do so. In particular, users can build their own class objects, which are better suitable for requirements of concrete Legion application like high performance or high security.

And finally, "Core objects implement common services". Legion defines the interface and basic functionality of a set of core object types that support basic system services, such as naming and binding, and object creation, activation, deactivation, and deletion. Core Legion objects provide mechanisms that classes use to implement policies appropriate for their instances. Examples of core objects include hosts (processors), vaults (data stores), contexts and binding agents (global naming systems agents), and implementations (system-specific code executives).

PropelXbi and Legion

It is desirable to envisage deployment of PropelXbi on Legion system to enable it function in multi-location Grid system. Unfortunately, Legion system was bought by a commercial organisation called Avaki and is no longer available to the public. Avaki doesn't expose Legion architecture anymore, but instead it offers three complete software Grid products which use Legion architecture as its base infrastructure. The offered products are Avaki Data Grid providing secure wide-area access to data stored on multiple locations, the Avaki Compute Grid, providing wide-area access to available remote processing resources based on business policies defined locally or centrally and the Avaki Comprehensive Grid which bundles the two previous products. The product which would be of interest to us is the Avaki Computational Grid, as it provides the functionality which we would leverage in the Legion architecture. Regrettably, no technical documentation is provided by the Avaki company and thus it is not possible to devise how PropelXbi could possibly be integrated with the Legion system.

Globus

The Globus Toolkit is a community-based, open architecture, open-source set of services and software libraries which support Grids and Grid applications (Foster & Kesselman 1999; Foster, Kesselman & Tuecke 2001; Globus Project 2001; Foster et al. 2002a; Foster et al. 2002b; Globus Project; Shalom 2002a).

The toolkit addresses issues of remote job submission and control, secure file transfer, system and service information, Grid security, information discovery, fault detection, resource and data management, communication and portability. Services of Globus Toolkit are accessible by simple well-defined APIs (for C and Java) hiding the underlying hardware and software heterogeneity. Services operate in Open Grid Services Architecture (OGSA) – a conceptual framework for grid computing formed by Globus research team.

Globus architecture – OGSA

Open Grid Services Architecture is a conceptual service-oriented model for Grid computing integrating Grid technologies and Web services (Foster et al. 2002b). In OGSA, Web services framework is further refined focusing on features required by Grid infrastructure and applications.

Via standard interfaces and conventions, OGSA supports creation, termination, management, and invocation of stateful, transient services as named, managed entities with dynamic, managed lifetime.

OGSA's fundamental concept is the adoption of a common representation of computational and storage resources, networks, programs, databases and the like. All are treated as services – network enabled entities that provide some capability through exchange of messages. More precisely, every entity is treated as a Grid service, which is a Web service conforming to a set of conventions that define how clients interact with a Grid service and provide for a controlled, fault resilient and secure management of a service. These conventions are represented by interfaces defined in Open Grid Services Infrastructure specification (OGSI) (OGSI-WG 2003), which has to be implemented by a Web service in order to be a Grid service. OGSI interfaces are defined in WSDL (Web Services Description Language) leveraging its language independency and the fact that this Web services standard is already broadly used and supported by industry. As these services expose their interfaces defined in language-independent WSDL, they can be written in any language for which a WSDL binding exists (e.g. Java, C, Python ...)

By defining this architecture, OGSA defines uniform exposed service semantics, identifies standard mechanisms for creating, naming and discovering transient Grid service instances, provides location and implementation transparency and supports integration with underlying native platform facilities. As the service interface is separated from its actual implementation, the service can have multiple implementations on different platforms which can take advantage of native facilities available on individual systems.

Thanks to this service-oriented abstraction model, OGSA enables consistent resource access across heterogeneous platforms. Moreover it provides a common framework for Grid services allowing inter-Grid operability, which was missing before. As another merit, services abstraction allows composition of complex services from lower-level services without regard to how these services are implemented.

Globus implementation – Globus Toolkit

As OGSA is fairly recent invention, there are two versions of the Globus Toolkit. Globus Toolkit 2 was written before OGSA/OGSI was published and therefore it doesn't incorporate OGSA architecture, but rather uses services commonly available in operating systems without any underlying unifying service model. Globus Toolkit 3 aims at implementing OGSA architecture but at the moment (April 2004) in current available release (3.2), some Globus services are OGSI-compliant already and some are still using the same basis as in previous versions. The intended migration from non-OGSA to OGSA architecture of Globus Toolkit is depicted on following picture.

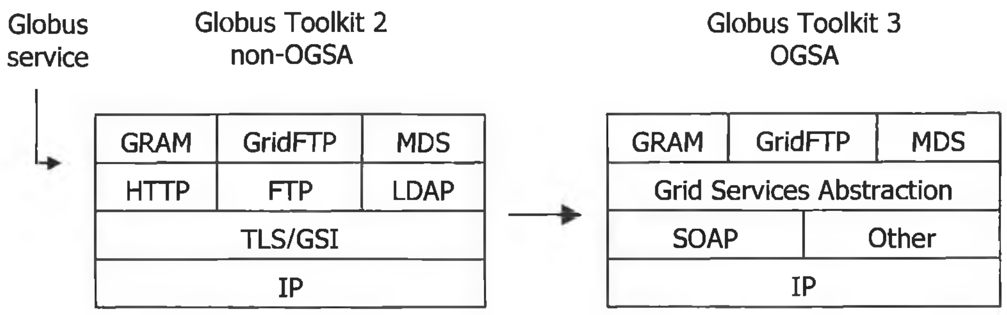


Fig. 7.15 non-OGSA to OGSA Globus transition

Note: GRAM (job submission service), GridFTP (data transfer service) and MDS (information service) are services provided by Globus Toolkit. TLS is transport-level security protocol (follower of SSL) and GSI is Grid security infrastructure.

It is expected that the OGSA architecture will be widely adopted throughout the Grid computing world and that over two years, the majority of grid systems will be OGSI-compliant with non-OGSI application being gradually abandoned (Globus Project).

PropelXbi and Globus

Globus Toolkit is written in C, but there are ways how to access its functionality from other languages as well. It is done through Community Grid Kits (CoG) for various languages. The latest stable Java CoG Kit (version 1.1) is compatible only with Globus Toolkit 2.4 and thus if we wanted to use Globus' functions directly in PropelXbi we would have to choose Globus Toolkit 2.4.

Globus Toolkit 2.4 is a set of services and as such it provides a good foundation layer for Grid computing. However, it is not concerned about job-scheduling and load balancing issues which are the domain of other higher-level grid schedulers. These schedulers use Globus services as its execution base and examples of them are Condor-G and AppLeS. These will be discussed in the following sections. The following figure shows the composition of the envisaged application – an assembly consisting of an application wrapper, a scheduler and Globus services.

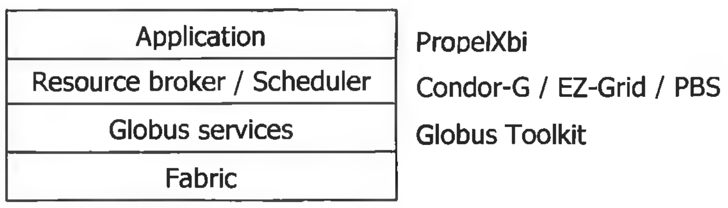


Fig. 7.16 PropelXbi on Globus

Schedulers

Globus and Legion provide a basic infrastructure for Grid computing taking care of distributed resource management. These two systems usually aren't used alone, but work as a base for higher-level (also called application-level) schedulers, which take care of job management. It is important to mention that not all application-level schedulers use Globus or Legion infrastructure, but several use their own means of resource management.

There are two types of application-level schedulers differing by the goal they strive to achieve. They are High Throughput Schedulers (HTS) and High Performance Schedulers (HPS). High Throughput Schedulers (also called Batch systems) want to achieve highest possible throughput (number of completed tasks/jobs) in given time, expecting non-ideal computing circumstances. Computing time is usually long, in order of days or weeks.

On the other hand, High Performance Schedulers aim at maximising performance of one individual application, expecting ideal computing circumstances and not caring about performance of other jobs running on the same system.

In short, High-throughput schedulers (HTS) are used for processing of massive amounts of data (in order of petabytes) over long periods of time (days/weeks) using any computer available. In contrast, High-performance schedulers (HPS) are used for jobs that aren't aimed at processing large amounts of data but need to be processed as quickly as possible and are usually run on special dedicated machines.

The other way to differentiate schedulers is by the question, which a user asks when considering their task. In the case of High-throughput applications a user asks "How many jobs can be processed in given time using all available machines?", whereas High-performance scheduler users ask "How fast can be this job done on this machine?".

There are many schedulers available today with various advantages and limitations. In following sections we will review these which are realistically utilisable in PropelXbi (according to their hardware and software limitations). Reviewed schedulers are Condor / Condor-G, Janet, Frugal, EZ-Grid, GRB, OpenPBS / PBS Pro, Platform LSF, N1 Grid Engine / Sun Grid Engine Enterprise Edition, Weblogic clustering facility and AppLeS.

High Throughput Schedulers

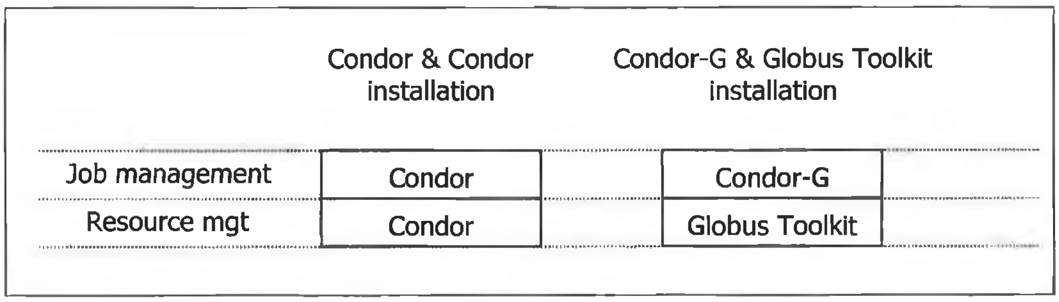
Condor

Condor is high-throughput scheduling system developed at University of Wisconsin (Condor; Condor-G; Bent & Thain 2002; Foster 2002; Frey 2002; Livny 2002; Condor Team 2003a; Condor Team). The Condor system consists of two parts. First part, called "Submit machine" takes care of job management (submitting jobs, keeping track of their status, gathering information about execution progress ...). The second part ("Execution machine") performs resource management, meaning that it controls resource availability and allocation.

Job requests, on submit side, and available resources, on execution side, are matched by Condor's matchmaker using concept of ClassAds (advertisements), which is similar to the advertisement concept used in JXTA. Every resource advertises itself, specifying its properties and optional usage restrictions (e.g. use only when load<5% and machine idle for 15 minutes). At the same time, every submitted job is also represented by a classad specifying the resources it needs. The matchmaker matches 'compatible' classads and informs the engaged sides of successful match. The matched entities then interact with each other independently without any intervention (or help) from the matchmaker's side.

As Condor is divided in two parts, it's not necessary to install both of them and only one can be installed if desired. The submit part allows submission jobs for execution and execution part carries out actual execution of a job. Both parts are usually installed, but it is not required.

The Condor system takes care of job and resource management, but because of the separation of its functions it also allows alternative mode of operation. Users which have Globus Toolkit already installed on their machines and would like to use Condor's job management capabilities can install Condor-G (Condor-G). Condor-G performs the job-management part of Condor and is specifically designed to use the Globus Toolkit as a resource managing base. The two possible operation scenarios are depicted on following figure.



	Condor & Condor installation	Condor-G & Globus Toolkit installation
Job management	Condor	Condor-G
Resource mgt	Condor	Globus Toolkit

Fig. 7.17 Possible Condor installations

The advantage of the Condor-G & Globus Toolkit installation is that users of already existing Globus Toolkit (GT) infrastructure can use it as before and additionally, they can use Condor-G job submitting facilities as well. Condor-G facilitates work with jobs of multiple tasks which are otherwise tedious to handle on plain GT. However, compared to the Condor & Condor installation, the Condor-G & Globus Toolkit doesn't provide job migration, process checkpointing and dynamic resource selection.

Both installations provide the main Condor merits – high-level job management, fault tolerance and credential management (automatic remote logging, authorisation and authentication).

The current version of Condor (6.6.3) runs on Linux and Windows NT/2000 machines. As the main developing effort is targeted on Linux environment, the current Windows version is missing several non-critical features. Compared with Linux, it doesn't have DAGMan (work-flow manager), doesn't support checkpointing and doesn't support access to files on shared network drives (condor automatically transports them to local drive). Condor-G is implemented only for Linux and uses the Globus Toolkit version 2.2 (non-OGSI compliant).

Janet

Janet (Capello; Capello 2003b) (formerly JICOS) is a research project from the University of California, Santa Barbara (UCSB) and is focused on developing a Java-based network computation system. It builds on experiences of systems previously implemented on UCSB - Javelin and CX, which are being cited in relation to distributed computing as well.

Janet acts as a scheduler, distributing work to host nodes being implemented by individual Java Virtual Machines. Its programming model is based on the concept of abstract distributed machine, which to a user, seems like a single computing machine, while it uses multiple execution nodes, which can be placed on different machines, hardware architectures and operating systems.

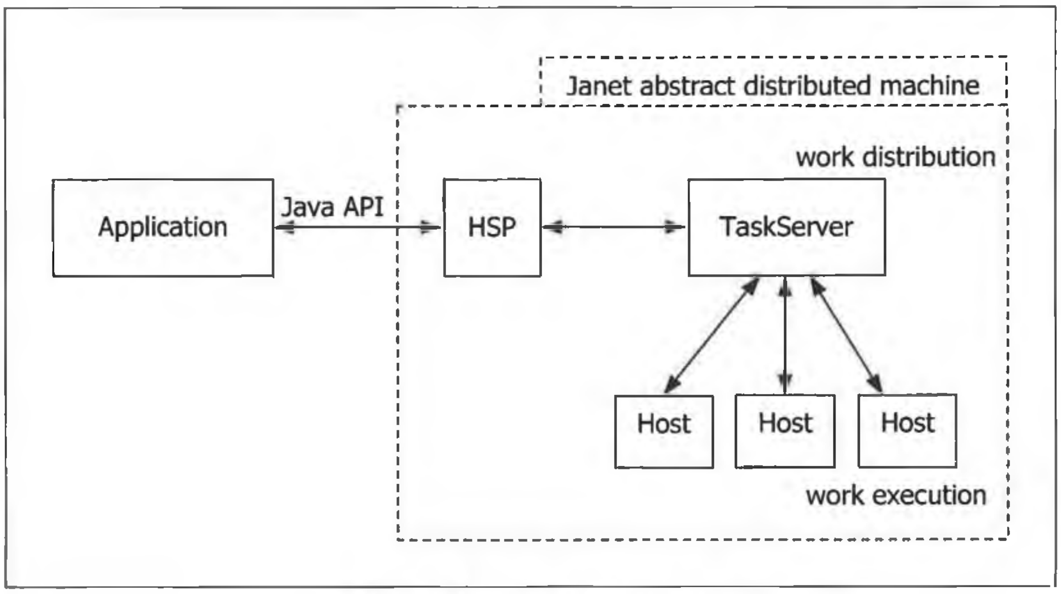


Fig. 7.18 Janet architecture

The abstract distributed machine consists of a Hosting Service Provider (HSP), TaskServers and Hosts. HSP serves as a point of contact between application and distributed Janet system. TaskServers take care of work distribution to individual Hosts, which are expected to be volatile – i.e. are expected to dynamically connect and disconnect from TaskServers. TaskServers moreover provide fault-tolerance feature by keeping track of assigned work and re-assigning work of these Hosts, which became unavailable.

Tasks executed by Host can spawn successor tasks or another sub-tasks and thus further decompose problem being solved and allow for greater parallelisation of computation and better use of parallel resources.

The Janet system is based on Java language and RMI/JINI calls between dispersed parts of the system. Java through its Java Virtual Machine provides a homogeneous platform on top of otherwise heterogeneous sets of machines and operating systems. This solves the problem of many other distributed systems, which fight with high heterogeneity of hardware and software configurations of execution machines.

Even though it would be desirable, Janet API doesn't provide the possibility of asynchronous task execution. A task can be executed in either synchronous mode, where the calling application waits until it receives result of a distributed computation or pseudo-asynchronous mode, where application spawns multiple tasks without waiting for their outcome and then at some point waits for result from all of them.

The synergy of PropelXbi and Janet can be imagined as suggested by the generic architecture of Grid-enabled PropelXbi Fig. 7.14.

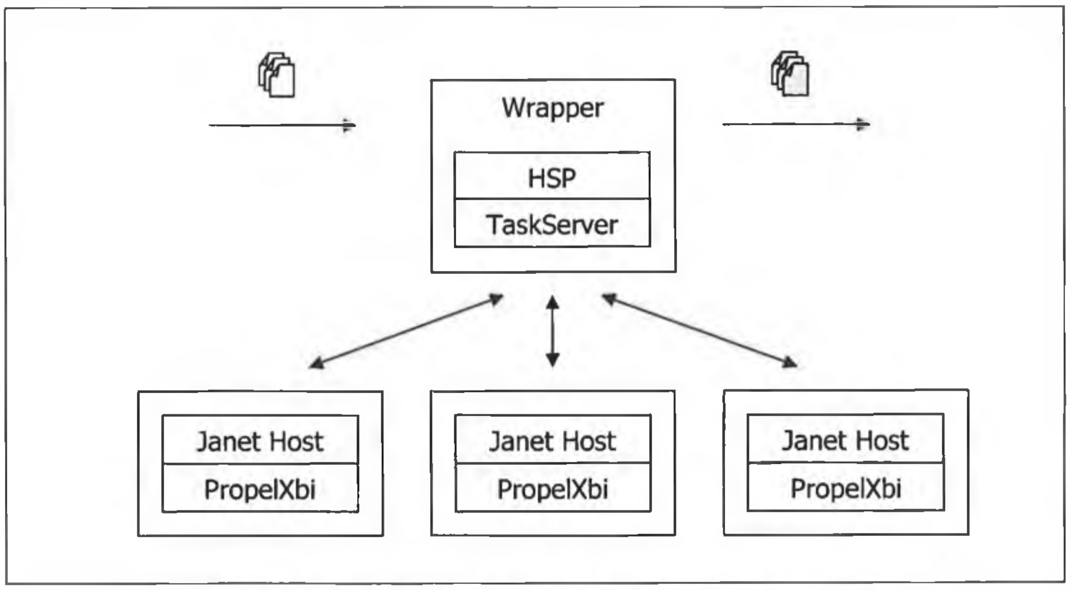


Fig. 7.19 Distributed PropelXbi using Janet system

As Janet's functionality is accessed through Java API, an application wrapper would need to be written, which would collaborate with Janet's Hosting Service Provider and a Task Server. Each computing node would then host the PropelXbi installation and a Janet Host, which would submit documents to a local PropelXbi installation. Janet by itself doesn't provide asynchronous calls, and thus additional code would need to be written to accommodate PropelXbi's intrinsic asynchronous behaviour.

Janet is a research project and at the moment it shows signs of un-robust behaviour and several bugs were encountered when experimenting with the current release (1.6.1). Even though it provides the functionality required for PropelXbi expansion to distributed computing, it still seems not mature and robust enough for considering its fully-fledged application in PropelXbi.

Frugal

Frugal is another distributed computing research project developed by R. Sean Borgstrom from John Hopkins University (CNDS; Borgstorm 2000).

It is built on Java and JINI technology. The system consists of Frugal Managers and Frugal Resources. Resources are JINI-enabled objects, registered in a JINI lookup service, residing on different machines which provide the execution power. Frugal Managers then control collections of Resources and when client asks a Manager to perform some work, the Manager selects the appropriate Resources and passes their reference to the Client. The Client then communicates directly with Resources.

Managers use a sophisticated strategy to select Resources, so that the overall CPU and memory load of the whole collection of computing nodes is minimized. The computing distribution strategy is called Differential PVM Strategy and in essence it selects that node, whose increase of load is minimal after assigning a given job to it.

The Frugal system was completed in 2000 and it seems that it wasn't updated since. In functionality tests, some parts of system were proven not to work with the current JINI release. As Frugal is based on obsolete JINI distribution, it can't be realistically used with PropelXbi.

EZ-Grid

EZ-Grid is research project form University of Huston (EZ-Grid; Chapman, Sundaram & Thyagaraja 2002). EZ-Grid sits on top of standard Globus Grid computing infrastructure (pre-OGSA version) and provides simple job submission interface hiding Grid computing internals from the user. Apart from user interface, EZ-Grid provides a job-scheduling broker, enhanced information service (more rich than default Globus information set) and offers means for richer definition and control of usage policies set on computing resources.

As mentioned above, EZ-Grid uses the infrastructure of Globus toolkit, which is accessed through Java CoG library – a Java interface to Globus services. Apart from indirect access to Globus, EZ-Grid also interacts directly with local schedulers in order to get additional detailed information not provided by Globus.

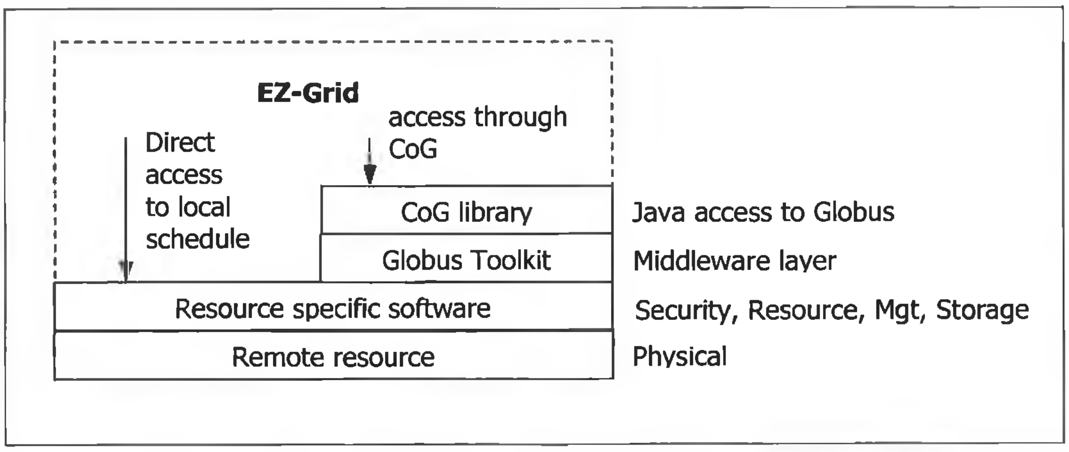


Fig. 7.20 EZ-Grid high-level structure

EZ-Grid Internal architecture is similar to other high-level Grid computing systems.

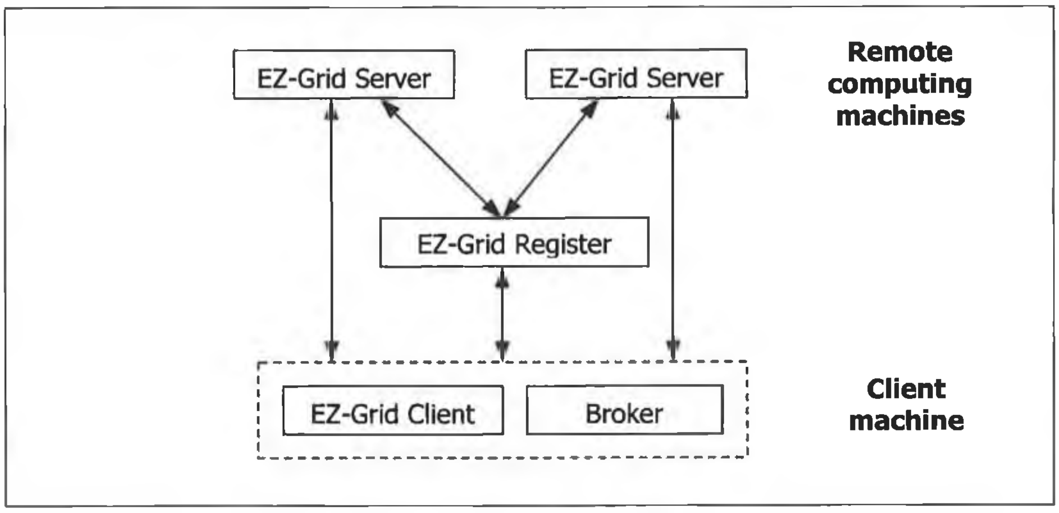


Fig. 7.21 EZ-Grid internal architecture

EZ-Grid consists of four principal components. Grid Server, Grid Register, Broker Kernel and Grid Client.

The Grid Server is located on every machine connected to the Grid and manages application profiles, job submission history and static and dynamic information about local resources. The Grid Register serves as central information server providing information about all available resources. As resources are often added and removed, it performs automatic resource discovery and periodically checks if registered resources are still available. The Broker Kernel performs matching between job requirements and available resources using sophisticated methods to achieve optimal load and time constraints. The last component, the Grid Client takes care of job submission through GUI and authentication and authorisation.

Even though we tried to contact EZ-Grid development team, we received no response from them and thus we cannot state what platforms are supported by EZ-Grid and how feasible is its incorporation in PropelXbi.

PBS

The Portable Batch System (PBS) (Altair Grid Technologies) is a batch queueing and workload management system originally developed for NASA. It operates on networked, multi-platform UNIX and Linux environments, including heterogeneous clusters of workstations, supercomputers, and massively parallel systems.

Every resource in PBS system is maintained by a PBS resource monitor. Resource monitors are used by job schedulers, which are in turn used by PBS servers. PBS provides job submission system either through GUI or command-line interface, keeping track of job progress, job priority and security management and job scheduling meeting various resource usage policies and load constrains.

PBS exists in two versions – freely available OpenPBS for non-commercial use (Altair Grid Technologies) and commercial PBS Pro (Altair Grid Technologies).

OpenPBS has basic features mentioned above, with a simple round-robin work distribution algorithm. In contrast, commercial PBS Pro offers more sophisticated work distribution algorithm, achieving better dispersion of work across different machines, better scalability and increased fault tolerance. PBS Pro furthermore provides support for Mac OS-X and Windows 2000 and XP platforms, support for cooperation with Globus Toolkit and an application programming interface.

Grid Resource Broker

The aim of Grid Resource Broker (GRB) project (Aloisio et al.; Aloisio et al.) developed in the HPC Lab of University of Lecce, Italy, is to create a simple GUI, which would allow trusted users to create, use and maintain Globus computational grids.

In order to use GRB, the user has to apply for an account at HPC Lab and has to have Globus 2.0 running on his computer. His own computational resource and other resources registered in Globus grid are then accessible through a Web browser, which provides the GUI interface. As GRB's only functionality is providing a GUI interface to Globus infrastructure it's not usable in PropelXbi. The only way to use GRB is through manual interaction with the graphical interface, which is not possible to use it in runtime code. If we wanted to use Globus infrastructure, it would be more pertinent to use the Globus Toolkit itself, anyway, and access it through Java CoG Kit API.

Platform LSF

Platform LSF (Platform Computing 2003d; Platform Computing 2003c) is a software tool for managing batch workload processing of compute and data-intensive applications. It allows scheduling of batch workload across a distributed, virtualised IT environment, utilizing all IT resources available including desktops, servers, supercomputers and mainframes regardless of their operating systems. Platform LSF runs on wide range of operating systems, covering Windows 2000, XP, various Linux and Unix flavours, Mac OS and supercomputer operating systems. Jobs are submitted either through a Web Browser, a command-line interface, an API or a SOAP interface.

Platform LSF is part of a Grid computing suite developed by Platform Computing (Platform Computing; Platform Computing 2003d). Other relevant Platform products are Platform JobScheduler (Platform Computing 2003b), which extends LSF's functionality with the Graphical design studio, where a business process containing intensive computing can be designed and a Control console for monitoring of scheduling and execution of batch processes is also provided. Platform ActiveCluster focuses on the utilization of unused computing cycles of Windows workstations. Platform Clusterware manages the entry-level batch application workload processing on Linux clusters. Platfrom MultiCluster manages resource sharing between multiple autonomous geographically spread LSF grids with differing local access policies.

The synergy of PropelXbi and Platform suite can be envisaged as illustrated by Fig. 7.14 with PropelXbi installations being satellites, providing computation power and Platform LSF doing the job of work scheduling and load balancing. In addition Platform JobScheduler can be used as a monitoring facility, providing on-line information about state of the work execution and includes an exception handler, which reports failures in processing and alerts the appropriate people of such events. In addition, Platform ActiveCluster can be augmented to harness the power of idle Windows desktop computers, if such enrichment is desired.

Platform also offers another product which is focussed on a different usage scenario – Platform HPC, which is aimed at enabling High Performance Computing, leveraging specialized high performance network interconnects of clustered systems or supercomputers. As it is a product aimed at High performance computing, its objective is to provide maximal application performance using all available hardware, usually over a short period of time. This product would be usable in situations where PropelXbi receives occasional computationally intensive tasks and its aim is to process them individually as quickly as possible. Its strength lies in its utilization of specialised network interconnects, which aren't always available on common systems. Supported operating systems are Linux and supercomputer systems.

N1 Grid Engine, Sun Grid Engine Enterprise Edition

N1 Grid Engine 5.3 (N1GE) (Sun Microsystems 2002a) (formerly Sun Grid Engine) and Sun Grid Engine Enterprise Edition 5.3 (SGEEE) (Sun Microsystems 2001b) (formerly Sun ONE Grid Engine, Enterprise Edition) are two distributed resource management software solutions allowing transparent use of distributed computing power (Sun Microsystems 2001a; Aberdeen Group 2002; Sun Microsystems 2002c).

The front-end development for both N1GE and SGEEE is done in the Grid Engine open source project (gridengine.sunsource.net) sponsored by Sun Microsystems. Sun does not deviate from the source code produced via the Grid Engine project for releases of N1GE/SGEEE. Reference releases, which are functionally identical to N1GE and SGEEE at a point in time, are available via the Grid Engine project. N1GE and SGEEE are both made from the same source tree in the Grid Engine project and share internal components. When Sun decides to release a new version of N1GE and SGEEE, it brings a stable build of Grid Engine software into the Sun quality assurance process and documents and offers the software under the N1GE/SGEEE brands.

Sun has a vision of various types of Grids differing by their size and span.

- Cluster Grids, which are grids dedicated to one project within one department.
- Enterprise Grids (or Campus Grids), which span multiple departments within one enterprise and can be used for multiple simultaneous projects.
- Global Grids, which go behind enterprise boundaries with resources shared over Internet.

N1GE provides functionality necessary for Cluster Grids computing, SGEEE then for Enterprise computing. Global grid needs are addressed by Globus

Toolkit, which is supported by Sun Microsystems as a partner of Globus development team.

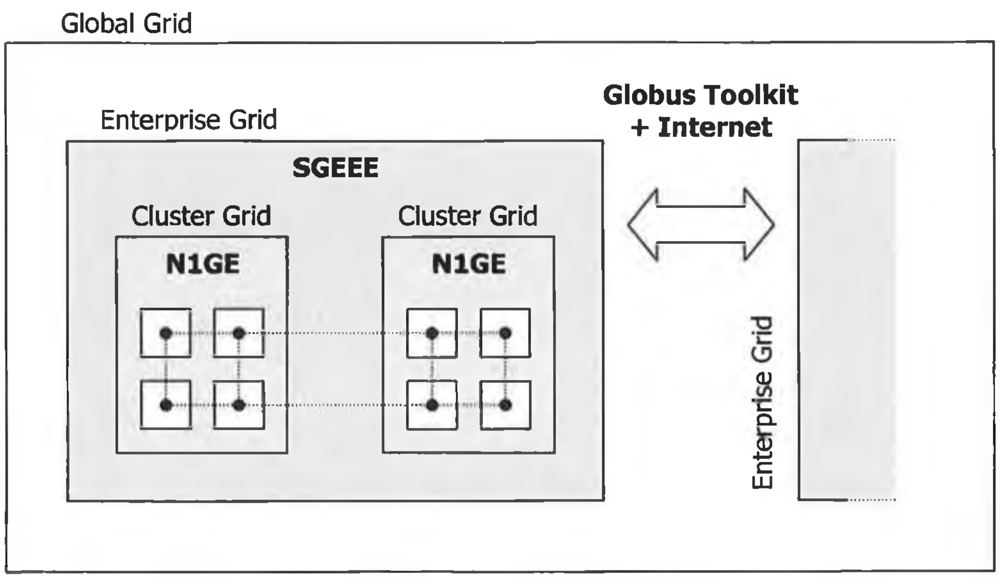


Fig. 7.22 Types of Grids according to Sun

The basic function of N1GE, is providing transparent access to all departmental resources by matching available resources in a grid with users' requests. N1GE supports both batch jobs, without need of user intervention, and interactive jobs for which it opens X-terminal window.

When a user wants to submit a job, he specifies a requirement profile for the job along with the user identification and a priority number. The requirements profile contains attributes associated with the job such as memory requirements, operating system required, etc. According to the profile and priority, N1GE then dispatches the job to a suitable queue associated with an appropriate host server on which the job will be executed. N1GE uses load-balancing techniques to spread the workload among available servers. To obtain necessary resources for execution, N1GE uses policies to examine available computational resources within the grid and allocates them to jobs in a manner that optimises their usage across the cluster grid. As N1GE is layered above the operating system it requires no alterations to applications.

N1GE runs on Linux and Solaris platforms and is free for personal and commercial use.

SGEEE provides all the functionality of N1GE, and in addition, it provides mechanism to allocate Grid computing resources based on policies, which dictate how resources are distributed among projects and people, not jobs. These policies are a level above job resource allocation. Incorporating these high-level policies allows SGEEE consolidate multiple cluster grids into enterprise grids, where multiple projects are run simultaneously and computing power is distributed according to firm's business objectives.

With SGEEE, a user, team, department, or project can receive a resource allocation for a period of time, based on some percent of the total resources available. SGEEE will ensure that the assigned percentage of resources is available to the jobs within that project or for a user, team, or department. SGEEE policies are flexible, so users and project teams can negotiate resource assignments that can vary from week to week.

SGEEE runs on Linux and Solaris platforms and is priced depending on grid size.

The standard version of N1 Grid Engine looks ideal for symbiosis with PropelXbi, as its job distribution feature is exactly what PropelXbi Grid extension requires. Individual documents can be passed as individual jobs, which are distributed on hosts containing PropelXbi installations, achieving load distribution and shorter processing time. N1GE, furthermore, has a favourable feature of being completely free for personal and commercial use. The only drawback of N1GE is that it runs only on Linux/Solaris platforms and thus potential existing Windows computers can't be used.

It appears that in relation to PropelXbi there isn't a need for the implementation of high-level policies, provided by Sun Grid Engine Enterprise Edition and the functionality provided by N1 Grid Engine is sufficient.

WebLogic clustering facility

At present PropelXbi is deployed either on BEA WebLogic application Server (BEA 2002a; BEA 2003a; BEA 2003c; BEA 2003b) or JBoss application server (JBoss). Even though WebLogic Server is not aimed at area of Grid computing, it provides the clustering mechanism, which in fact can do the work of the surveyed Grid schedulers. The currently used distribution of JBoss (3.0.5) doesn't provide any JMS clustering facility, while it is promised to be implemented in future release of JBoss 4.0. For that reason we focus only on clustering features of the WebLogic application server.

A cluster is a group of servers, which appear to user as a single "super" server, in same way as machines in Grid appear to user as single "super" computing machine. The difference is that, members of cluster group are of the same platform and mostly with the same operating system and software facilities. Furthermore, computers in a cluster are more tightly bound, often requesting LAN connection (and thus not allowing connecting over Internet), as clusters installations are not meant to exceed institutional boundaries.

Weblogic provides a feature called "distributed destinations" (also called virtual destinations) which allows PropelXbi's extension to distributed computing. A distributed destination is a set of physical destinations (places where JMS messages can be sent) called under a single JNDI name, so they appear to be a single logical destination to a client. Each member of such set can be placed on a different machine in a cluster and must be placed in a separate JMS server.

When a message is sent to a distributed destination, a load-balancing algorithm is used to choose to which particular member of a set is the message redirected, so that messages are evenly spread over all members and the overall load is optimised. Available load-balancing algorithms are plain round robin, random scheduling and weighted variants of both mentioned algorithms, where weights assigned to individual destinations determine which destinations are more preferred (and thus receive more load).

In addition, distributed destinations provide a fail-over feature. When one member becomes unavailable due to server failure, the traffic is redirected to other available members in the set.

An interesting fact considering load balancing in general, is BEA's statement about why they didn't implement more advanced load balancing algorithms than round robin and random. They come from observation that a standard application server work load contains many short-running requests. They state that in this setting, parallelism is most efficiently exploited by processing each request on as few servers as possible, as the overhead for communication is relatively large. The consequence is that simple round robin or random load balancing schemes are particularly effective and it is rarely worth the effort either to take actual server load into account or to redistribute on-going work when it occasionally becomes unbalanced. (BEA 2003a)

This statement is interesting because the situation of PropelXbi is almost identical, with timespans of processing of individual documents being of rather small to medium length.

High Performance Schedulers

AppLeS

AppLeS used to be High-performance scheduling project from University of Carolina, San Diego. As the objectives of original development team broadened, AppLeS project led to the establishment of GRAIL laboratory with various projects covering different aspects of Grid computing (GRAIL). GRAIL projects, which would be relevant to our research, are GrADS (Dail) and AMWAT (AMWAT).

The goal of the GrADS project (Grid Application Development Software) is to enable development and performance tuning of Grid applications by simplifying distributed heterogeneous computing. This aim is meant to be achieved by providing a set of C libraries, hiding the details of low-level Grid programming from users. The GrADS project is currently in development stage.

AMWAT (AppLeS Master Worker Application Template) aims to do similar thing as GrADS, but is focused on Master-Worker (divide and conquer) scenario. It is a C library that makes it easier for programmers to develop applications that solve a problem by breaking it into subproblems, distributing the subproblems to multiple processes (typically running on multiple CPUs), and combining the subproblem results into an overall solution. The AMWAT library takes care of scheduling, communications, and fault tolerance, allowing the developer to concentrate on the application-specific aspects of the program. In contrast with GrADS, AMWAT's implementation exists already.

As both these projects produce C libraries, they are not suitable for amalgamation with PropelXbi, as it is all written in Java.

Most suitable candidates for PropelXbi

Considering PropelXbi scenario it is difficult to identify the most suitable scheduler. PropelXbi works as a document transformation engine, taking documents in, transforming them and outputting them afterwards. Under this prospect, processing of each document can be viewed as a separate job, which can be individually scheduled. As there are two different ways in which PropelXbi can be used, there are also two different performance metrics leading to two different performance goals.

The first possible PropelXbi use is as a transformation engine for a large set of documents (of possibly large size) that need to be modified or transformed from one format to a new one (e.g. converting data from a legacy database to XML format). The goal here is to achieve the highest possible throughput of data rather than maximised speed of processing of individual documents. In this case, frequency of incoming documents is high (as they are most probably loaded straight from local disk). Most suitable schedulers for this scenario are the High-throughput schedulers.

The second possible use of PropelXbi is as an on-the-fly transformer of messages between two or more applications. In this case, the performance metric is speed of message transformation and thus suitable type of schedulers are the High-performance schedulers. The size of the transformed documents is rather small or moderate and frequency of their arrival is also of rather small to moderate.

Scenario characteristics and fitting schedulers are summarised in following table.

PropelXbi use	Data size	Doc. arrival frequency	Goal	Suitable scheduler	Scheduler example
Large data transformer	Large	High	High throughput	**HTS** - High throughput	Condor, N1 Grid Engine
On-the-fly transformer	Small/ Medium	Low/ Medium	High speed	**HPS** – High performance	AppLeS, Platform HPC

Tab. 7.5 PropelXbi use scenarios and suitable schedulers

However, most of available schedulers, which title themselves “High Performance Schedulers”, are focused on support of high performance versions of C, C++ and Fortran languages. PropelXbi is written in Java and cooperation with such schedulers would be rather difficult, resulting in cumbersome solutions, trying to overcome language differences, rather than utilising their intrinsic advantages.

Nevertheless, as PropelXbi usage scenarios are not likely to be of extreme nature, High Throughput Schedulers may possibly do the same work with satisfactory results.

Following table summarises the surveyed high throughput schedulers and their relevant features:

Name	Vers	Submit	Res. Layer	Lang	Origin	Cost	Distrib	Platf
Condor	6.4.7	CLI	Condor	Java	Res	Free	Rules	Win, Lin
Condor-G	6.4.7	CLI	Globus	Java	Res	Free	Rules	Win, Lin
Janet	1.7	API	Janet	Java	Res	Free	Plain	Win, Lin
Frugal	-	API	Frugal	Java	Res	Free	Diff PVM	Win, Lin
EZ-Grid	-	?	Globus	Java	Res	Free	Rules	???
GRB	-	GUI	Globus	-	Res	Free	???	Lin, Unix
OpenPBS	2.3	GUI	PBS	?	Com	Free *	Plain	Lin, Unix
PBS Pro	5.3	API GUI	PBS, Globus	?	Com	Com	Rules	Win, Lin, Unix, Mac
N1 Grid Engine	5.3	API CLI GUI	Sun	C	Com	Free	Rules	Lin, Unix
Platform LSF	5.1	API CLI GUI SOAP	Platform	?	Com	Com	?	Win, Lin, Unix, Mac

Sun Grid Engine Enterprise Edition	5.3	API CLI GUI	Sun	C	Com	Com	Rules	Lin, Unix
WebLogic clustering facility	-	-	Web Logic	Java	Com	Com **	Plain, Rand.	Win, Lin, Unix

Tab. 7.6 High throughput schedulers

* - commercial use is prohibited

** - included in installation of BEA Weblogic Server, on which are some PropelXbi installations currently deployed

The table first specifies name and version of surveyed scheduler. Then it specifies how jobs are passed to the scheduler – through use of API, from command-line interface, from GUI or by passing a SOAP message. The Res. Layer column refers to the software package used in the resource layer, as some products rely on different software packages for handling resource issues. Lang specifies the programming language that the software is written in. The Origin refers to the origin of the product – either as the research project or the commercial product and the cost column states whether it is freely available or priced. The Distrib column identifies the algorithms used for the load distribution – plain round robin, random, differential PVM or more sophisticated rules. The Platf column states on which platforms the software runs.

We divide our assessment of suitable candidates into two halves – first we consider the candidates which are freely available and then the commercially available candidates.

Among the freely available schedulers, there are two competitors for the best choice – N1 Grid Engine and WebLogic clustering facility. N1GE has advantage of more sophisticated scheduling algorithms than WebLogic. Its disadvantages are that it is written in C, and so job submitting must be done through command-line interface external to Java and the only supported platforms are Linux and Unix, not allowing use of existing Windows computers. WebLogic clustering facility isn't free, but as it is part of WebLogic server, on which some PropelXbi installations are currently deployed, it can be considered to be so if it is such case. WebLogic has the advantage that the clustering facility is native to the platform on which PropelXbi runs and that load-balancing can be implemented by mere configuration of the WebLogic cluster without the need of any changes in the way PropelXbi currently works. The drawback of WebLogic is that it provides only simple round robin and random scheduling. However, it has been stated that efficiency of these scheduling algorithms is sufficient. If the deployment application server is JBoss and there isn't need to use Windows computers, then N1 Grid Engine is the clear choice. In the other scenario, where the deployment application server is WebLogic, the best scheduling device is WebLogic's native clustering facility.

In the group of commercially available schedulers, Sun Grid Engine Enterprise Edition is salient with its unique feature of sophisticated people/project centred resource allocation policies not available in any other product. If such feature is not needed then other schedulers provide comparable functionalities. In addition, other schedulers may be considered if Windows and Mac computers need to be used, as SGEEE is the only commercial product, which does not support them. In such cases, the selection would be based on product pricing and brand preferences.

CHAPTER 8

SUMMARY AND APPRAISAL OF SURVEYED TECHNIQUES

8 Summary and appraisal of surveyed techniques

In this chapter we first briefly review current XML pipeline processing model as represented by the XPipe paradigm's implementation PropelXbi in section 8.1. In section 8.2 we then review the techniques we surveyed in chapters 5 to 8 and point out how their particular features can enhance the current XML pipeline processing implementation. In last section 8.3 we summarise the identified potential enhancements of PropelXbi, present the enhancements we decided to implement and briefly describe them.

8.1 Current XML pipeline processing implementation

The current technical implementation of the XML pipeline processing system was described in detail in Section 1.2. Before we progress to surveyed techniques, we summarise the current architecture.

Documents, to be transformed, are placed in JMS messages and sent to a JMS queue. The JMS queue serves as a storage space for documents being processed. Above the JMS queue there is pool of MDB objects, which observe contents of the queue and when a document appears there, one of the MDBs retrieves it from the queue, loads the appropriate transformation component (XComponent) with which it is to be transformed, transforms it and returns it back to the queue. As MDBs are self-sufficient objects, they are automatically managed by the server, which takes care of their whole life-cycle management. The fact, that MDBs load the appropriate XComponents dynamically, allows them to be assigned to places in the pipeline with the current highest workload. After being processed by all stages of the pipeline, the document is removed from the processing queue and placed in a separate storage space designated for fully transformed documents.

8.2 Appraisal of surveyed techniques

The following paragraphs, list the surveyed techniques and point out how their particular features can enhance current XML pipeline processing represented by PropelXbi as implementation of the XPipe paradigm.

8.2.1 Parallel processing

Our study of parallel processing revealed four enhancements, which are, however, already present in PropelXbi.

The first is the concept of pipeline processing. Processing is divided into individual stages, which can be executed in parallel. This is implemented by XComponents, which are independent components of whole document transformation.

The second is the technique of instruction pre-fetch and caching. In this technique, instructions which were recently used or which are likely to be used soon, are kept in memory, avoiding the need for lengthy access to the physical memory. As the size and number of the XComponents is many times less than the size of available memory, all the XComponents are loaded in on start-up and during execution there is never a need to access the physical memory to load them.

The third revealed technique is data forwarding. Consecutive units, which process the same data, pass intermediate results directly to each other, saving time which would be otherwise spent by saving intermediates to a storage space and loading them in again. In PropelXbi case, this can be utilized only when the processing times of the involved XComponents are short, because otherwise it would hinder the parallelism feature currently present in PropelXbi. PropelXbi implements the data forwarding concept using the XComponent compiler, which is discussed in greater detail in next section about Jackson Inversion.

The last concept from area of parallel processing is vector pipeline chaining. This trick is used where two consecutive vector units process the same vector of data. The second unit connects directly to the output of first one and starts processing already finished vector elements even before all remaining elements are finished by the previous unit. In PropelXbi, the technique of vector pipeline chaining is implemented by the Scatter/Gather components. When document contains a group of independent elements, it is divided by the Scatter, and each piece is processed independently of the rest. By such doing, the Scatter in fact transforms a solid document into a vector of its independent parts. Thanks to this element separation, parts of the document can be processed by later stages even though some other parts were not yet processed by stages placed earlier in the transformation pipeline. When all elements of original document are processed, the Gather component assembles them together into the final document.

8.2.2 Jackson Inversion

The concept of Jackson Inversion is to transform a set of programs, which communicate with each other through temporal storage spaces into one monolithic code, which incorporates all the individual programs. Communication between the former individual programs is then implemented by function calls from one block of code to another. The aim is to simplify and speed-up the whole programme compound. This transformation can be applied to a set of XComponents, but as mention earlier, their processing time span has to be short, so that loss of parallelism is negligible. Sections 6.3 and 6.4 present two designs based on Jackson Inversion. The first is an on-line XComponent compiler, where the decision on which components to compile is made by PropelXbi's inner logic during the run of the transformation. The second is the off-line XComponent compiler, where the components to compile are selected manually by the user before the transformation runs.

8.2.3 Distributed Computing

In area of distributed computing, we researched three topics, where each has potential for PropelXbi enhancement.

TupleSpaces

TupleSpaces come with the concept of a global distributed space to which objects are written and from which they are read. The innovative aspect is that all issues of space distribution are hidden from user and that a very simple interface is used for objects manipulations. Using TupleSpaces, programmers can very quickly develop applications performing distributed computing.

In PropelXbi, TupleSpaces can be used to replace the JMS queue currently functioning as document storage space. As TupleSpaces can be easily distributed, such replacement would facilitate shift of PropelXbi from one-machine to distributed computing. However, conversion to TupleSpaces would require a change of the transformation mechanism as well. It seems that the cost of work that would need to be spent on such a change would be higher than benefits of PropelXbi's distribution. Besides, such benefits can be gained more easily by using Grid technologies discussed in a later paragraph.

Project JXTA

Project JXTA is a set of language independent peer-to-peer communication protocols. An enhancement, which can be brought by using JXTA protocols, is that it can be used as an internal communication mechanism for distributing PropelXbi's work. Client then could be written in any language for which there is JXTA binding (currently Java and C) and it would be possible to use a wider range of machines and transforming devices to do PropelXbi's work.

However, there are numerous disadvantages of using JXTA as PropelXbi's internal communication mechanism. JXTA communication is unreliable, lacks persistence mechanisms and hinders a load-balancing feature which is natively present when using JMS communication system. Furthermore, by implanting JXTA into PropelXbi we would be futilely replacing a native communication mechanism, which is innately available in the current PropelXbi architecture.

Grid Computing

Grid computing comes with the possibility to use a group of physically distributed machines as one big computing device. Various grid products take care of work distribution, process monitoring and collection of results. High throughput and computing power can be achieved by a using grid scheduler to distribute documents to different machines, so that they are processed in parallel.

8.3 Identified potential enhancements and selected implementations

Our research has identified several potential enhancements, which can be utilised for streamlining PropelXbi. The first two are on-line and off-line XCompilers. XCompilers increase processing speed by compiling components together. They remove the need to spend time on saving documents to intermediary storage space. An on-line XCompiler decides which components to compile in run-time whilst in the off-line XCompiler case, this decision is made by the program users. The next enhancement is provided by TupleSpaces which provide a potential replacement for the JMS queue. Such a substitution would facilitate the expansion to distributed computing. However, the same goal can be achieved more conveniently with Grid technologies. The JXTA Project offers an alternative communication system, which is independent of the underlying hardware and operating system. Yet, its unreliability and lack of a persistence mechanism, hinder its employment and in addition, the communication system currently present in PropelXbi is wholly sufficient anyway. The Grid technologies offer performance enhancement by providing facilities to distribute document processing on multiple machines. By such, documents can be processed in parallel, which results in shorter total processing time.

From studied techniques and enhancements, we chose to implement the Off-line XCompiler and a Grid-based distributed version of document processing system. Having considered technical complexity and time constrains, these two were identified as the most promising in terms of potential performance improvement and technical feasibility.

The first implemented enhancement is an Off-line XCompiler. XCompiler compiles XPipes into self-contained transforming devices called compiled pipelines. Compiled pipelines are the building blocks of the compact J2SE-based runtime of PropelXbi.

The second implemented enhancement is the Grid-based distributed document processing system. The distributed version uses features provided by the Condor package to distribute documents on multiple machines where they are processed by compiled pipelines.

Discussion of implementations with evaluation of how big benefit they actually deliver is subject of Part 3 which follows.

PART 3

DOCUMENT PROCESSING

ENHANCEMENTS IMPLEMENTATIONS

CHAPTER 9

XCOMPILER

9 XCompiler

In chapters 6.3 and 6.4 we developed concepts of On-line and Off-line XComponent compilers (XCompiler). An XCompiler is a program, which compiles multiple XComponents of a pipeline into one self-contained transformation package. The difference between on-line and off-line versions is, that in an on-line XCompiler, it is the computer who decides which segments of XPipe to compile, whereas in the off-line case, it is the programmer or user, who chooses which XComponents to compile.

The high-level concept and technical implementation of an off-line XCompiler are discussed in the following sections 9.1 and 9.2. In section 9.3 we examine the performance improvement brought by an off-line XCompiler and the causes of PropelXbi's worse performance. We decided to implement the Off-line XCompiler. This will be referred to by the shorter term as XCompiler from now on.

9.1 Concept of XCompiler

The XCompiler transforms a pipeline of XComponents into one self-sufficient package (called a compiled pipeline), where all the code and information necessary for the execution of multiple XComponents is collocated into a single location.

This transformation eliminates the need to save and load intermediate results, which are passed between XComponents, as they can be held in memory and passed directly to next XComponent. This change provides a significant performance improvement, as accessing a permanent storage medium is usually the most lengthy part of computer transformations.

Another benefit of the XCompiler is possibility to run transformations through XPipes without the need of running the whole PropelXbi engine, where the start-up and run time demand a lot of resources and take a relatively long time. The runtime execution engine of compiled pipelines is based on simple Java class invocation, whereas PropelXbi transformation engine is built on Enterprise Java Beans. The former proves to be more efficient which is demonstrated and discussed in section 9.3.

9.2 Technical implementation of the XCompiler

We have designed and implemented the XCompiler as a Java program consisting of two parts. The first is the XComponents compiler and the second is the COmpiled PiPeline Execution Runtime (Copper). This division comes from the two stages in which the XCompiler is used. The first is compilation of the pipeline and the second is an execution of the document transformation by Copper.

9.2.1 Compiler

The task of the XComponent compiler is to create a self-contained package implementing the transformation defined by a given XPipe and its respective XComponents. Compilation, in this context, means producing a device which performs the same transformation as its defining sources (XPipe and XComponents), but independently of them, without any need to access them. The objective of the compiler is that the generated code must be executable by the standard Java Virtual Machine without need of J2EE environment.

The XCompiler achieves decoupling from J2EE environment by creating a package of standalone Java classes representing the transformation code of each XComponent. In addition to Java classes, an XML file is created which contains information about each XComponent, its type, parameters and required libraries.

Compiler architecture

Following figure Fig. 9.1 shows compiler's architecture:

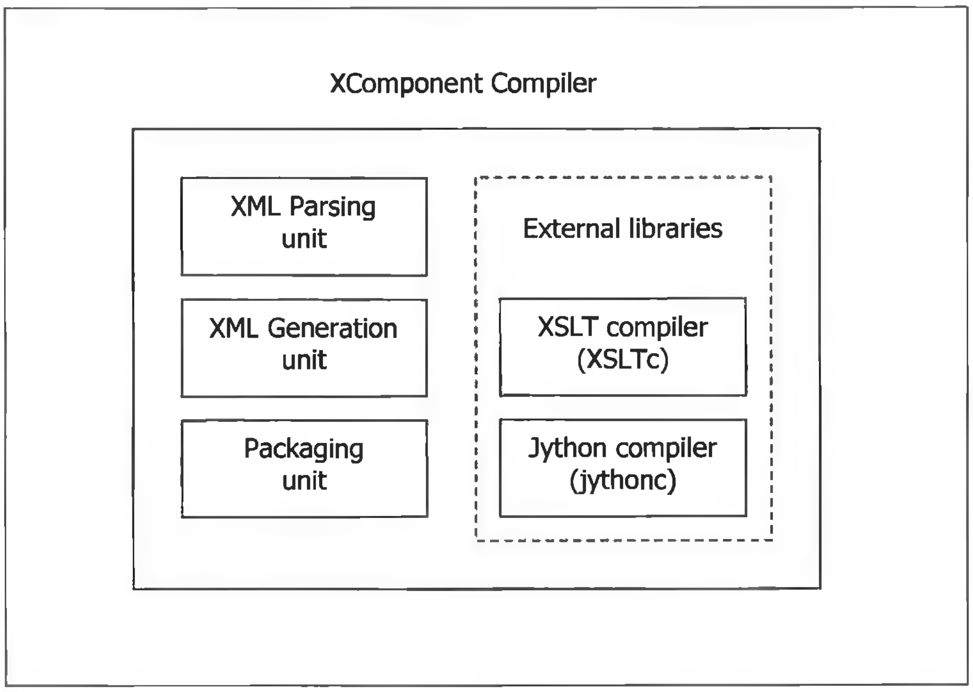


Fig. 9.1 Architecture of XComponent compiler

The compiler consists of three essential internal units, taking care of parsing XML documents, generating XML output and packaging of generated code. For the compilation purpose, two external libraries are used. The first is publicly available XSLTc (Joergensen 2001) for compilation of XSLT sheets. The second is freely available jythonc (jythonc) for compilation of Jython scripts.

The actual work of the compiler is illustrated in Tab. 9.1, and is further described in following text:

```
1) Create temporal folder for generated code

2) Parse XPipe definition file;
   for each XComponent {

 a)      Parse XComponent definition file;
 b)      Acquire XComponent transformation code;
 c)        case (XComponent is Java):
             copy Java class
           case (XComponent is Jython):
             compile code with jythonc
           case (XComponent is XSLT):
             compile code with xsltc
           case (XComponent is Exec):
             copy command definition
  d)     Copy required libraries for execution code
  e)     Add record to command list with information
         about XComponent (type, parameters, libraries)
    }

3) Copy run-time execution classes (Copper)

4) Package generated code and command list into
   executable JAR archive
```

Tab. 9.1 Conceptual code of compiler

The compiler first creates a temporary folder into which all the generated code is copied. At the end of the compilation process, all data in this temporary folder is packaged into a JAR archive.

Next, the compiler parses the XPipe definition file. In PropelXbi, the XPipe is represented by an XML document containing list of its constituting XComponents and their parameters. The XComponents referenced by XPipe are defined in other XML files containing descriptions of their type, the transformation code they provide, their parameters and other information related to their execution. By parsing the XPipe definition, the compiler acquires a list of XComponents used in given transformation and a list of parameters supplied to individual XComponents. Following that, it performs standard compilation loop on each of used XComponents.

The compilation loop starts by parsing the XComponents definition file. If the compiler determines that an XComponent was processed in some of previous compilation loop, it skips the subsequent three steps and continues to the last one.

The next step of the compilation loop is the acquisition of the transformation code. XComponents can either specify the location of the external code (e.g. Java class file, an XSLT sheet etc.) or can have the code embedded in themselves in a special CDATA element. When the transformation code is embedded in the XComponent, it is either kept as plain text (which is case of XSLT sheets, Jython scripts and Exec commands) or as a binary file encoded by Base64 algorithm. Base64 algorithm encodes binary data using only 64 alphanumeric characters of standard ASCI encoding, so that it can be treated as a text. This encoding is necessary when embedding binary files into XML documents, as CDATA elements must not contain any data which aren't Unicode characters (W3C 2000). Depending on how code is referenced in XComponent, the compiler either loads the transformation code from disk or extracts it from the XComponent definition file.

The third step of the compilation loop is compilation of the acquired code. For each compiled XComponent, a new unique sub-folder is created in the temporal folder, so that code from different XComponents is clearly separated. All code resulting from this step is placed in its corresponding sub-folder. Depending on the type of transformation code, compilation can result in four different actions.

Java compilation

When transformation code is a Java class, it is simply copied into its sub-folder as Java classes are already compiled and no further compilation is required.

Jython compilation

Scripts written in Jython are essentially python scripts which are interpreted by Java Python interpreter (Jython). In order to create Java classes which would do the same work as the original Jython script does, it is compiled by a Jython compiler, jythonc. Jythonc produces a jar file which contains a Java class with the same name as the Jython script. This class then performs the same work as the Jython script would have done if it were interpreted by a standard Jython interpreter. Apart from the compiled class, the jar file also contains precompiled Jython libraries which are necessary for execution of the compiled script.

Jythonc also provides an option to exclude libraries aren't from the resulting jar. This would be beneficial if the Jython libraries were located on some know location and were reused for all compiled scripts. The advantage is that the resulting jars are smaller as the runtime libraries aren't included in each of them. However, as goal of XComponent compiler is to create a stand-alone transforming device, it must not rely on presence of any external libraries and thus this option could not be used.

XSLT compilation

XSLT sheets perform transformations of any XML files to which they are applied. One way of using them is to compile them into translets (transformation applications) first. A translet is a Java class which performs the same XML transformation as the original XSLT sheet would do. This translet is then either run from command line, or invoked at runtime as a normal Java program. The advantage of compiling XSLT sheets into translets is that execution of the Java program is faster than applying standard XSLT stylesheets.

This fits nicely into what we want to achieve and so we use XSLTc to compile XSLT sheets contained in XComponents into translets and use the generated translet code as the transformation code.

Exec compilation

Exec definitions are textual commands. At the time of execution, those commands are given to the underlying operating system to execute. Hence, no compilation is needed and the whole exec command is simply copied into an information record which is persisted in the last step of the compilation loop.

The next step of the compilation loop copies the libraries, which are necessary for execution of generated transformation code, into the transformation package.

The compiler provides an option to note the location of libraries without their physical copying. This leads to smaller size of the generated code, but makes compiled pipeline dependent on presence of required libraries on the classpath.

The last step of the compilation loop enters a record into the XML command list which contains runtime information about the processed XComponent. Namely, it describes the transformation code, states the XComponent's type, its parameters and libraries it needs for execution.

When all the XComponents are processed, the compiler adds command list and Copper classes (execution runtime) to the transformation package. As a final step, the compiler packages all prepared files into an executable Java archive (JAR) file.

The JAR file created by compiler (compiled pipeline), performs the same transformation as the XPipe from which it was created, but can be run stand-alone from command line which dramatically speeds up execution as shown in 9.3.

Usage example

To show how compiler works, lets have a look at an example of the compilation process. Suppose we have an XPipe called XPipe1, which consists of 5 XComponents as on following figure Fig. 9.2.

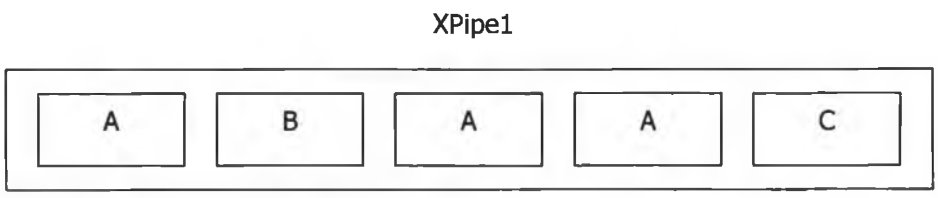


Fig. 9.2 XPipe example – transformation view

To illustrate different types of XComponents, let's say that XComponent A is a Java XComponent, B a Jython XComponent and C an XSLT XComponent.

Third and fourth components of XPipe1 are the same as XComponent A with different parameters.

In the file system, XPipe1 is implemented by following files:

```
\XPipe1.xpi  (XPipe definition file)
\A.xco       (definition file of XComponent A)
\B.xco       (definition file of XComponent B)
\C.xco       (definition file of XComponent C)

\A.class     (transformation Java class of XComponent A)
\B.py        (transformation Jython script of XComponent B)
\C.xslt      (transformation XSLT sheet of XComponent C)
```

Fig. 9.3 XPipe example – file system view

We have chosen to make all the XComponents reference their code externally, so that it better illustrates what happens in the compilation process.

When the compiler is run on XPipe1.xpi, it produces a compiled pipeline, contained in XPipe1.jar. Structure of XPipe1.jar is on following figure Fig. 9.4.

```
\1\A.class    (transformation code of XComponent A)
\2\B.jar      (transformation code of XComponent B)
\5\C.class    (transformation code of XComponent C)

\cmdlist.xml       (an XML file containing information
                    about how XPipe1 should be executed)

\Transform.class   (execution runtime class)
\*.class           (other necessary execution classes)
\lib\*.jar         (necessary Java libraries)

\META-INF\manifest.mf (JAR description file)
```

Fig. 9.4 Structure of example compiled pipeline

Folders 1, 2 and 5 contain generated transformation code. As XComponents 3 and 4 use the same code as XComponent 1, their transformation code isn't generated anew, but the code of XComponent 1 is reused. Both XComponents

A and C are compiled into Java classes, as they are of type Java and XSLT. The Jython XComponent B is compiled into a jar file.

Other files, which are needed for execution of the compiled pipeline apart from the transformation code, are also included in XPipe1.jar file. The cmdlst.xml is an XML file which contains information necessary for execution. The Transform.class and associated classes implement the actual execution runtime (Copper). The Manifest.mf is a file which gives information about the archive file and informs Java Virtual Machine that Transform.class should be invoked, when the compiled pipeline is run from the command line.

The cmdlist.xml for our example is shown on figure Fig. 9.5

```
<commandList>
    <xcomponents>
        <xco number="1">
            <type>JavaClass</type>
            <code dir="1" package="">A.class</code>
            <classpath absolute-classpath="true">
                C:\xml-apis.jar
            </classpath>
            <parameters>
                <param name="Param1">111</param>
            </parameters>
        </xco>
        <xco number="2">
            <type>Jython</type>
            <code dir="2" package="">B.jar</code>
            <classpath absolute-classpath="true">
                C:\jython_core_libs.jar
            </classpath>
            <parameters/>
        </xco>
        <xco number="3">
            <type>JavaClass</type>
            <code dir="1" package="">A.class</code>
            <classpath absolute-classpath="true">
                C:\xml-apis.jar
            </classpath>
            <parameters>
                <param name="Param1">222</param>
            </parameters>
        </xco>
        <xco number="4">
            <type>JavaClass</type>
            <code dir="1" package="">A.class</code>
            <classpath absolute-classpath="true">
                C:\xml-apis.jar
            </classpath>
            <parameters>
                <param name="Param1">333</param>
            </parameters>
        </xco>
        <xco number="5">
            <type>XSLT</type>
            <code dir="5" package="">C.class</code>
            <classpath absolute-classpath="true">
                C:\endorsed\xalan.jar;
                C:\endorsed\xsltc.jar
            </classpath>
            <parameters>
                <param name="newName">CCCC</param>
            </parameters>
        </xco>
    </xcomponents>
</commandList>
```

Fig. 9.5 Example cmdlist.xml

Execution runtime is described and example of how the execution is performed is given in following section.

9.2.2 Execution runtime (Copper)

The execution runtime (Copper) performs the document transformation which is embodied in the compiled pipeline. Figure Fig. 9.6 depicts Copper's architecture.

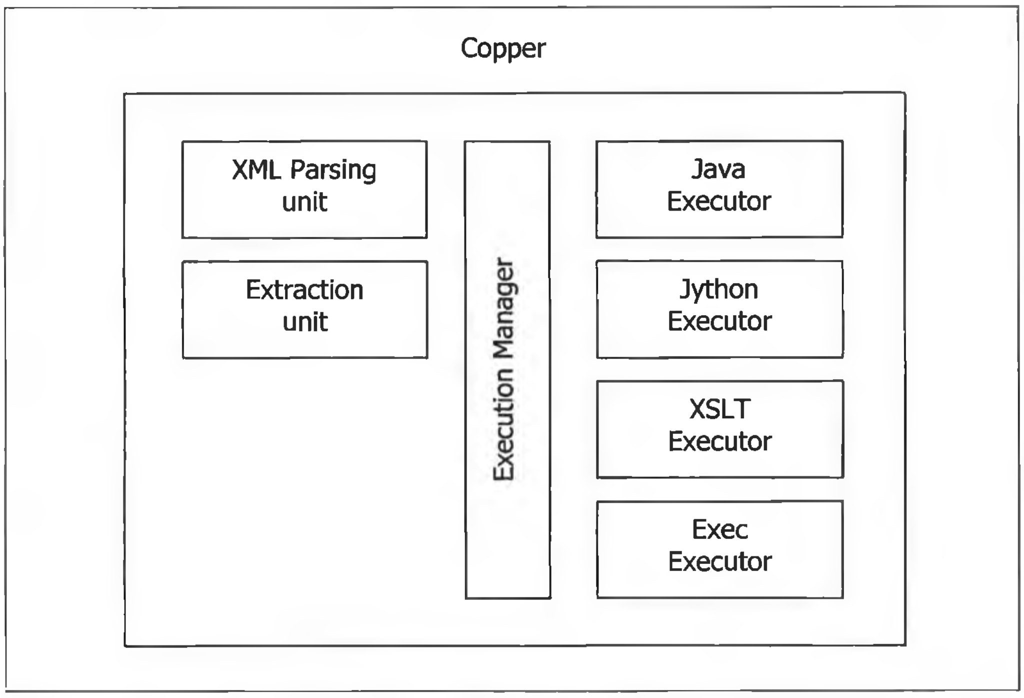


Fig. 9.6 Architecture of Copper

Copper consists of four essential units. XML Parsing unit is used for parsing command-list, which provides Copper with all the information necessary for correct execution. The Extraction unit is used for the extracting of data from the JAR files. The Execution manager takes care of the flow of document through the XComponents during the transformation process and the Executor classes perform the actual execution of transformation code.

Functioning of Copper is illustrated by following high level code:

```
1) Extract content of JAR archive into temporal folder

2) Parse command-list file;
   for each XComponent {

 a)     Acquire input document
 b)       case (XComponent is Java):
            Load libraries required by XComponent;
            Create instance of transformation class;
            Set instance's parameters;
            Execute transformation instance;

          case (XComponent is Jython):
            Load libraries required by XComponent;
            Create instance of transformation class;
            Execute transformation instance;

          case (XComponent is XSLT):
            if (correct Xalan is NOT available):
              Stop;
            Save input to temporary file;
            Create parameter list;
            if (correct Xalan is already in JVM):
              Create instance of transformation class;
              Execute transformation instance;
            else:
              Instruct OS to execute command which creates
              new JVM that runs the transformation;

          case (XComponent is Exec):
            Pre-process command definition for current OS;
            Instruct OS to execute command definition;

c)      if (occurred Error):
            save state of document before last XComponent;
            save Error message;
            Stop;
        else:
            if (next XComponent can read input from memory):
               keep result in memory
            else:
               save result to disk
   }
```

Tab. 9.2 Conceptual code of Copper

The pipeline execution, performed by Copper, starts with extraction of the content of the archive which contains the compiled pipeline. This is necessary, as the libraries needed for execution can't be loaded form an archive file.

Next step is parsing the command-list file, which contains information about how the process of transformation should be performed. The list of XComponents which should be executed is obtained and for each of them, the Execution manager performs following sequence.

Firstly, it either loads the input document from disk or acquires it from memory, depending on if previous XComponent produced result as a file or data kept in memory. Then, it passes document and information about transforming code to appropriate Executor, which carries out actual transformation of the document. Depending on the type of XComponent, execution can be run in four different ways.

Java execution

To be able to run given Java class, we first need to get its instance. This however, can be a difficult task, because not only we need to load the class, but also all libraries it uses. Therefore, the first step of the Java execution is loading the libraries referenced by the transformation code.

The Java executor has to follow the same algorithm as is used by Java Virtual Machine (JVM) to locate the libraries. First, it searches the system classpath, then it inspects the folder in which the transformation class is placed, then the \lib folder in the compiled pipeline archive and finally classpath which may be passed in from command-line. If the required library is found on any of those paths, it is loaded in the memory.

After all necessary libraries are loaded, an instance of the transformation class is created. If there are any parameters, which change the XComponent's behaviour, they are set by invoking appropriate methods of the created instance. The XComponent system requires that all Java codes implementing XComponent with parameters must have methods of type setXxx(String value), where Xxx is the name of parameter. Thanks to that, parameters can be set at run-time by knowing name of its setting method. So for example, if there is a parameter "count", it is set in the instance by calling its "setCount" method.

Furthermore, the XComponent system requires, that all Java XComponents must have an execute(in, out, err) method which performs the transformation. Thence, after all parameters are set, input, output and error variables are supplied to the execute() method and the method is invoked using standard Java invocation call.

Jython execution

Invocation of Jython code starts by loading the Jython specific run-time library and creating an instance of the transformation class. Unfortunately, because of the way that jythonc works, it is not possible to pass any parameters to compiled Jython scripts.

In all compiled Jython scripts used for XComponents, there is method called “main” which is the main (and only one) access point to script’s functionality. Transformation is then executed by invoking the “main” method of the instantiated class with input, output and error variables passed in as parameters.

XSLT (translet) execution

In order to run translets, XSLTc and the Xalan library, version 2.5 or greater, must be available to run-time of JVM. This however causes a lot of difficulty. As we want to remove the dependence of compiled pipelines on external libraries, we need to check if the correct versions of XSLTc and Xalan are available.

The XSLT Executor checks if Xalan is already available in current JVM. If it is not available, it checks the \lib folder of the compiled pipeline and the classpath supplied on command line to see if the correct Xalan and XSLTc libraries can be found there. If all checks are unsuccessful, execution stops, as it is not possible to continue with the transformation.

When it has checked that execution of translet can go ahead, the input document is saved to the temporal directory as translets take files as their input.

In next step, the Executor constructs a list of parameters which will be passed to the translet. Translets take parameters in a special format and the parameter list must be formatted accordingly.

As a final step, the Executor performs the execution which is different for systems where Xalan and XSLTc are already available to JVM and systems where Executor needs to use supplied Xalan and XSLTc libraries.

In the first case, Executor simply passes the parameter list to the "main" method of the class org.apache.xalan.xslt.Process and invokes it, which is the standard way of invoking translets. We need Xalan and XSLTc already loaded by the JVM as the class org.apache.xalan.xslt.Process is located in XSLTc library, which requires the correct version of the Xalan library for its execution.

The second case, where JVM doesn't have Xalan/XSLTc loaded or where it has the wrong version of Xalan, is more complicated. Xalan and XSLTc libraries can't be simply loaded into JVM, but have to be loaded as "endorsed libraries" (Sun Microsystems 2002b). This is achieved by passing special directive to JVM on its startup. In our case, we can't change settings of the JVM that is available. We have to create a command which starts a completely new JVM with a directive specifying the location of the Xalan and XSLTc libraries and stating that they are "endorsed libraries". This command is then passed to underlying operating system for execution with the org.apache.xalan.xslt.Process and the prepared parameter list as its arguments.

Exec execution

The Exec definition is a line of text presenting a command which is to be run by the underlying operating system. This text can contain three special words – "SOURCE", "OUTPUT" and "ERROR" which are replaced by the full names of the input, output and error files respectively. In addition, in Windows, names of files have to be surrounded by double quotes as they can contain spaces, which can cause malfunction of the system commands.

In order to be able to run the command specified in the exec definition, we have to prefix it with the name of executing command interpreter. In Unix systems, command interpreter is shell “sh”. In Windows 95 and 98, it is “command.com” and in newer versions of Windows it is “cmd.exe”.

The command specification prefixed with the interpreter is then submitted to the underlying operating system, which executes it.

When the Executors have finished executing the document transformation, the Execution Manager checks if the transformation has run correctly, or if there were any errors during the transformation. If an error occurred, the Execution Manager stops the transformation and creates two information files, which can help in determining what caused the problem. The first file contains the state of document before it was submitted to the last XComponent and the second is an error message which was received during its processing. If no error occurred, it either saves the resulting document to the disk, or keeps it in memory, if following XComponent can read its input from memory.

Example of execution

To illustrate how execution works, lets have a look at execution of the compiled pipeline XPipe1.jar, created in previous section. As the pipeline is encapsulated in an executable jar, it is invoked by the standard Java jar invocation command:

```
java -jar Xpipe1.jar in.xml out.xml error.xml
```

Using this command, we instruct the pipeline XPipe1 to transform in.xml and save the resulting document in out.xml. If an error occurs, the state of the document before entering the erroneous XComponent is saved in error.xml.

Process of execution of XPipe1 is shown on the following figure Fig. 9.7.

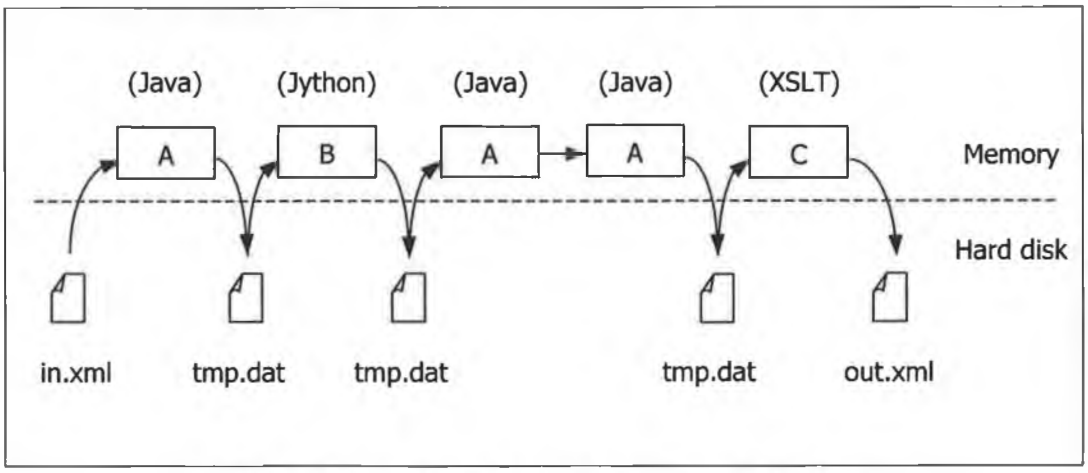


Fig. 9.7 Execution of example compiled pipeline

The execution example shows that documents are saved to temporal files between those XComponents that can't read or write its input and output to memory (Jython, XSLT and Exec XComponents). Conversely, the XComponents which can read and write documents to and from the memory pass the intermittent documents directly to each other (third and fourth components).

9.3 Performance Comparison

To evaluate the performance improvement provided by XCompiler, we carried out performance tests on the execution of pipelines compiled by the XCompiler and pipelines run in the current PropelXbi. First, we describe the testing procedure and the performed tests. Then, in section 9.3.1 we present the performance results we obtained and in section 9.3.2 we discuss the causes and reasons of PropelXbi's bad performance. Finally, in section 9.3.3 we draw conclusions and suggest ways of how to improve performance of current XML processing.

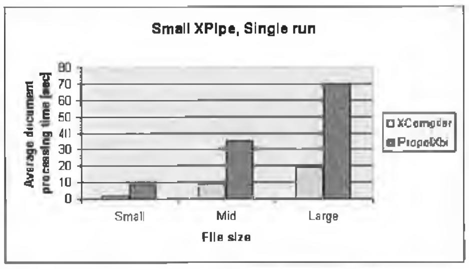
To test the transformation performance of the current PropelXbi implementation and the XCompiler, we carried out the following performance tests. We selected three sample XPipes each of different complexity (small, mid-size and large). For each pipeline we created three test files of small, mid and large size. Each test file was run through its pipeline in two scenarios. The first scenario was the transformation of single file, where the test file was consecutively submitted to the pipeline five times in a row, but at any given time, there was only one file being processed by the pipeline. The second scenario was the transformation under heavy load, where a group of twenty files was submitted to a pipeline in a batch. The average processing times stated in the following tables are calculated from the times for all processed documents obtained in their respective scenarios. All tests were run on Pentium III 1Ghz, with 512 MB of memory and Windows 2000 operating system.

9.3.1 Processing Performance Results

The first tested XPipe was a small pipeline of four Java XComponents, which carried out the transformations of CSV files to XML files. Sample files were of size 10 KB, 800 KB and 1600 KB. The tables and graphs below give the results of execution of compiled pipeline (XComp) and the pipeline in PropelXbi for each testing file size and each scenario. The last column is the ratio between the processing times of compiled pipeline and PropelXbi.

Single	XComp	P'Xbi	P/X
Small	1.6 s	8.2 s	513 %
Mid	9.2 s	35.4 s	385 %
Large	18.8 s	69.8 s	371 %

Batch	XComp	P'Xbi	P/X
Small	2.3 s	7.9 s	343 %
Mid	9.7 s	342.1 s	3527 %
Large	19.7 s	crash*	--- %



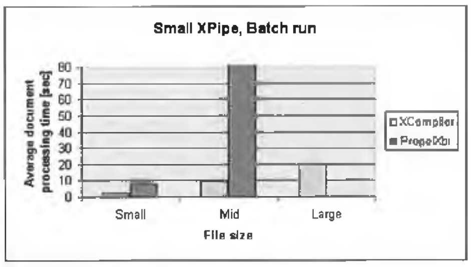


Fig. 9.8 Transformation performance of small-size pipeline

* PropelXbi crashed after processing 3 documents of 20. Reported problem was Out of memory error.

The tables and graphs show the performance results for a single and batch runs for small, mid and large size documents.

In the single run scenario, the execution of the compiled pipeline showed to be 3 to 5 times faster than the transformation by the current PropelXbi application. The slow-down of PropelXbi decreased with larger sizes of test files. We believe that, the main source of inefficiency of PropelXbi is the overhead associated with the maintenance of Enterprise Java Beans. This premise would explain why PropelXbi performs worse in runs of small files. In such runs, the time spent by useful activity – that is the transformation itself – is lesser compared with the overhead which stays relatively the same.

Batch run scenario showed the even greater inefficiency of the current PropelXbi architecture. When the number and size of documents transformed by PropelXbi increased, the EJBs started blocking each other and the overall time of transformation greatly grew. In the small-size file run, the transformation of individual files didn't overlap each other and thus processing time didn't change significantly. However, in the mid-size file run, the EJB overhead manifested itself in such a way, that PropelXbi performed 35 times slower than compiled pipeline. As another proof of the inadequacy of the EJB approach, the large-size file run didn't finish at all, as PropelXbi crashed after processing three files out of twenty. The reason give for the crash was shortage of memory, even with 512 megabytes available.

In the batch run, the compiled pipeline didn't exhibit any significant performance fluctuations as a batch processing is executed in the same the way as single file runs. By its construction, the compiled pipeline can process only one file at a time and thus if batch of files is submitted to it, successive files are not taken into processing until previous ones are finished. For the same reason, there aren't any problems with lack of memory resources.

The second XPipe tested, was a mid-size pipeline of 31 Java and 1 XSLT XComponent. Pipe was used for processing of a sample UBL Order, generating standard business response according to UBL Op65 schema. UBL (Universal Business Language) is a set of standardized XML business documents for automated business interoperation. The sample purchase files were of size 7 KB, 50 KB and 100 KB.

Single	XComp	P'Xbi	P/X
Small	9.2 s	38.1 s	414 %
Mid	11.5 s	40.6 s	353 %
Large	13.9 s	44.0 s	317 %

Batch	XComp	P'Xbi	P/X
Small	10.0 s	42.2 s	422 %
Mid	12.3 s	43.3 s	352 %
Large	14.7 s	47.0 s	320 %

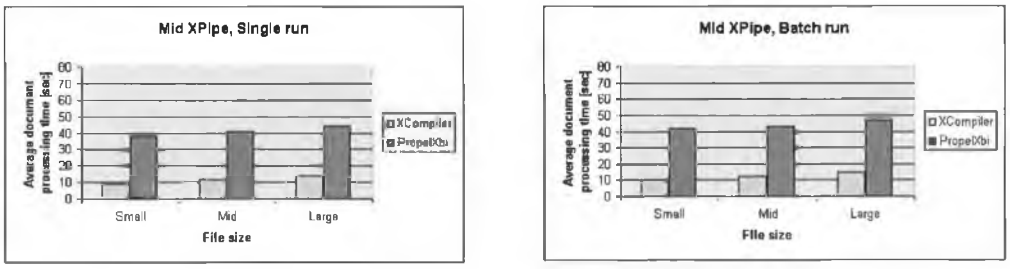


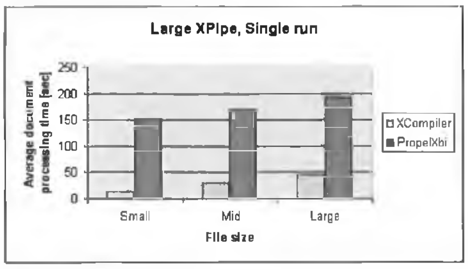
Fig. 9.9 Transformation performance of mid-size pipeline

In this test, the compiled pipeline again proved 3 to 4 times faster and the relative difference was again greater in small-size file runs as the actual useful processing time shortened in comparison with the fairly stable overhead time. In contrast with the previous test, the batch run processing times weren't immensely different from those of the single run. This is caused by the small size of files passed through the pipeline. In the previous case, the files were of sizes in order of hundreds of bytes, in this case, they were in sizes of order of tens of kilobytes. Because of that, the EJB architecture didn't consume such huge amounts of memory as in the previous case and the EJB server didn't get to a state of congestion.

The last XPipe tested, was a large-size pipeline of 87 Java XComponents. The large XPipe implemented partial conversion of a bill file from ccML (XML mark-up used for OpenOffice documents) to LexML (Legislation Mark-up Language). The sample bill files were of size 20 KB, 100 KB and 200 KB.

Single	XComp	P'Xbi	P/X
Small	14.4 s	151.4 s	1051 %
Mid	29.0 s	169.2 s	583 %
Large	45.4 s	192.2 s	423 %

Batch	XComp	P'Xbi	P/X
Small	15.0 s	173.4 s	1156 %
Mid	30.1 s	194.8 s	647 %
Large	46.2 s	221.5 s	479 %



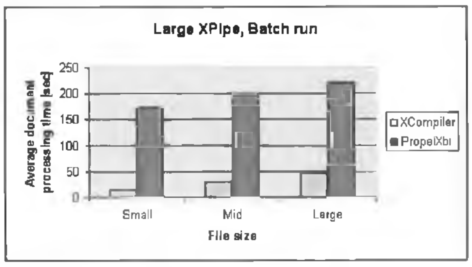


Fig. 9.10 Transformation performance of large-size pipeline

Similar to the previous tests, the execution of the compiled pipeline proved faster. In this case, the performance difference was even greater, with large files being processed by the compiled pipeline 4 times faster and small files even 10 times faster than by PropelXbi. Again, as in the previous test, the batch-processing scenario gave worse results for PropelXbi, due to contention of the EJB's for memory and processing time.

9.3.2 Processing Pattern Analysis

In this subsection, we look on the reasons of the poor performance of PropelXbi and assert observations about PropelXbi's performance in different conditions, namely in single and batch runs.

Processing Pattern Analysis

In order to examine the transformation performance of PropelXbi, we remind you how we defined document transformation phases in section 2.1, as we will decompose the document transformation into such stages. These transformation stages are shown on the following figure, which depicts the decomposition of the document transformation in a pipeline, consisting of two XComponents.

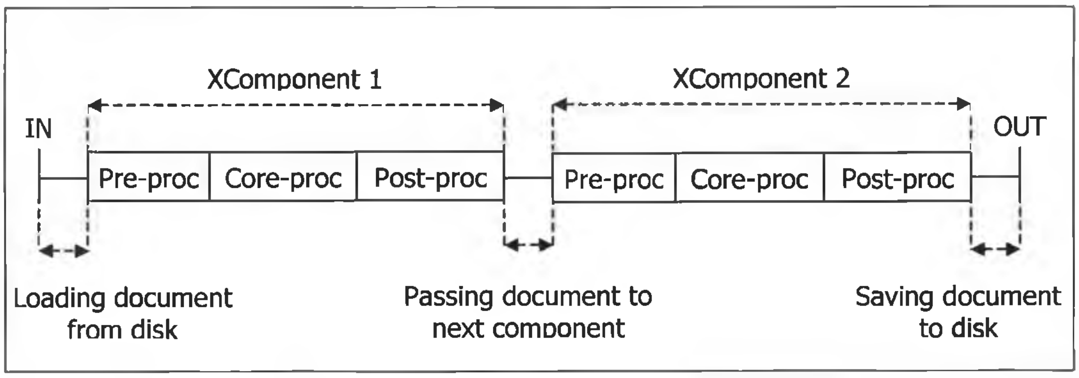
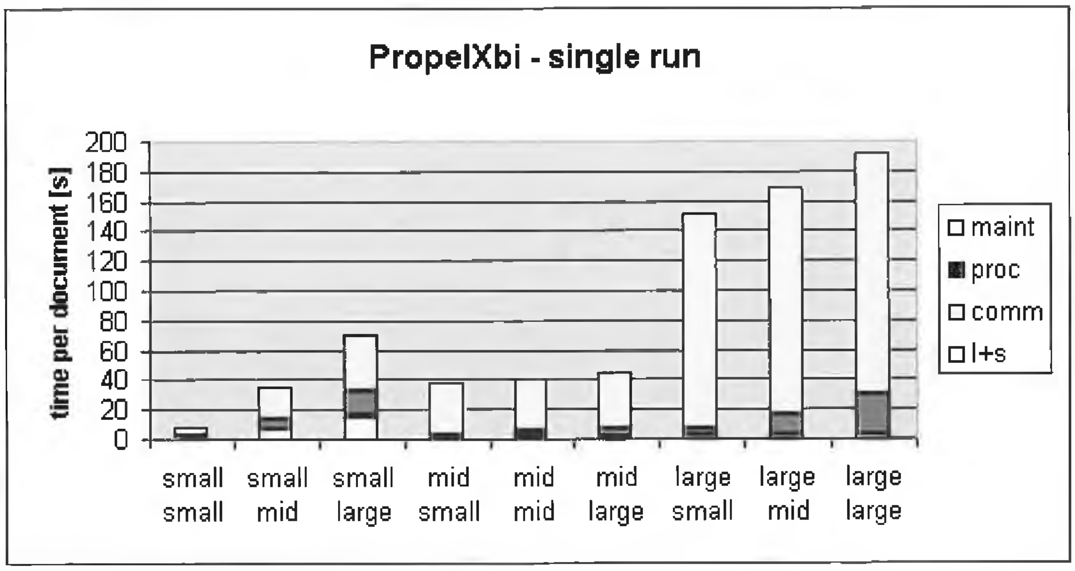


Fig. 9.11 Document transformation stages

The first step in the document transformation is loading the document from disk or acquiring it from any other storage, which serves as source of the documents to process. When it's loaded, it is passed to the XComponent, in which processing takes three phases. The first phase is the Pre-processing phase where preliminary actions take place. In case of PropelXbi and compiled pipelines, these preliminary actions are for example, acquiring runnable XComponent code, loading required libraries, setting XComponent parameters and checking pre-conditions. Unlike in the compiled pipelines, PropelXbi carries out one extra action, which is the extraction of the actual document from the received JMS message. The second phase is Core-processing phase which is the execution of the actual transformation which changes the content of the document. The last phase of in-component processing is the Post-processing phase in which the output of the document is further processed, post-conditions are optionally checked and in the case of PropelXbi, an output JMS message is assembled, which carries the transformed document to the next component. Pre-processing and Post-processing are considered to be maintenance stages, as the actual transformation is carried out only in Core-processing stage. Following the processing in one component, the resulting document is either passed to the next component, or saved to disk, if there aren't any other transformations needed.

When we look on the performance data with these stages in mind, we can instantly see the performance impediments in PropelXbi and their cause.



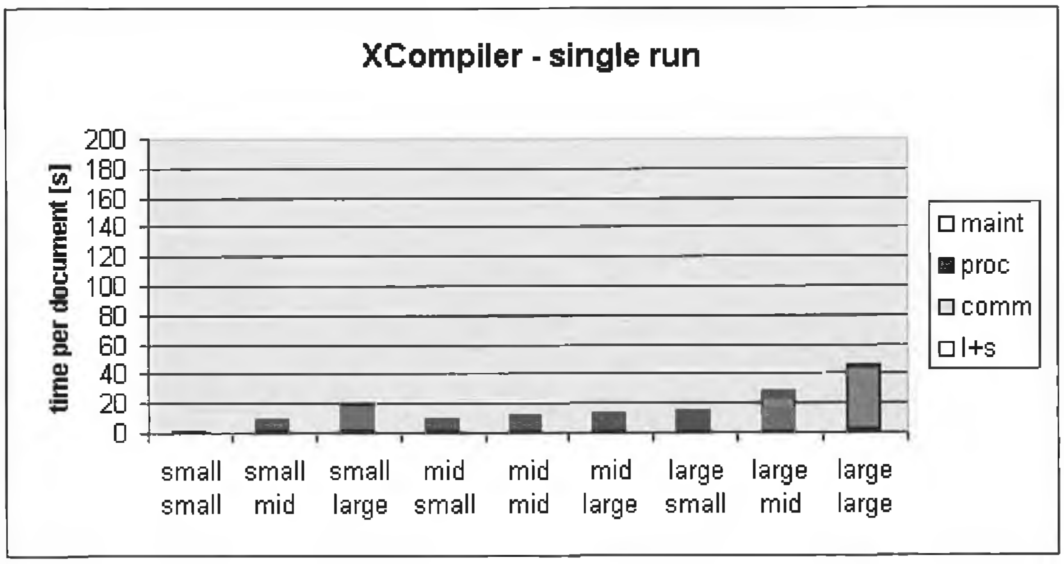
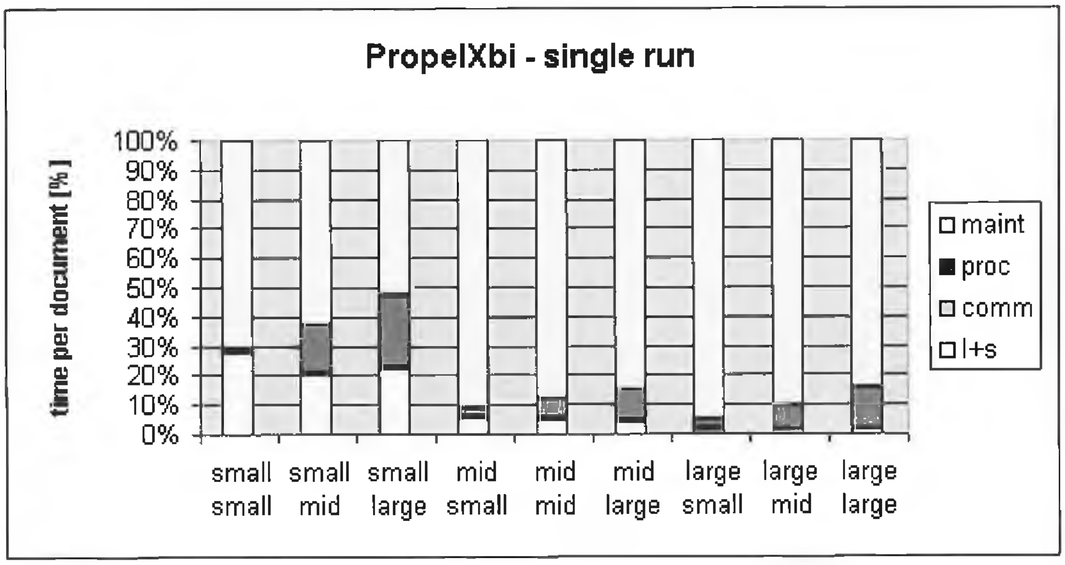


Fig. 9.12 PropelXbi and XCompiler transformation stages in single run



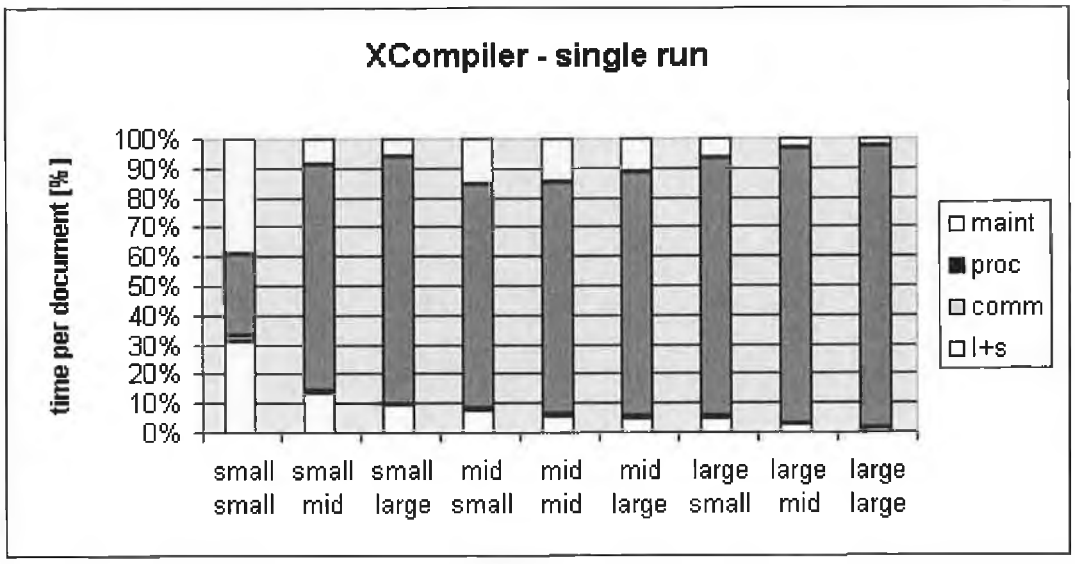


Fig. 9.13 Ratio of PropelXbi and XCompiler transformation stages in single run

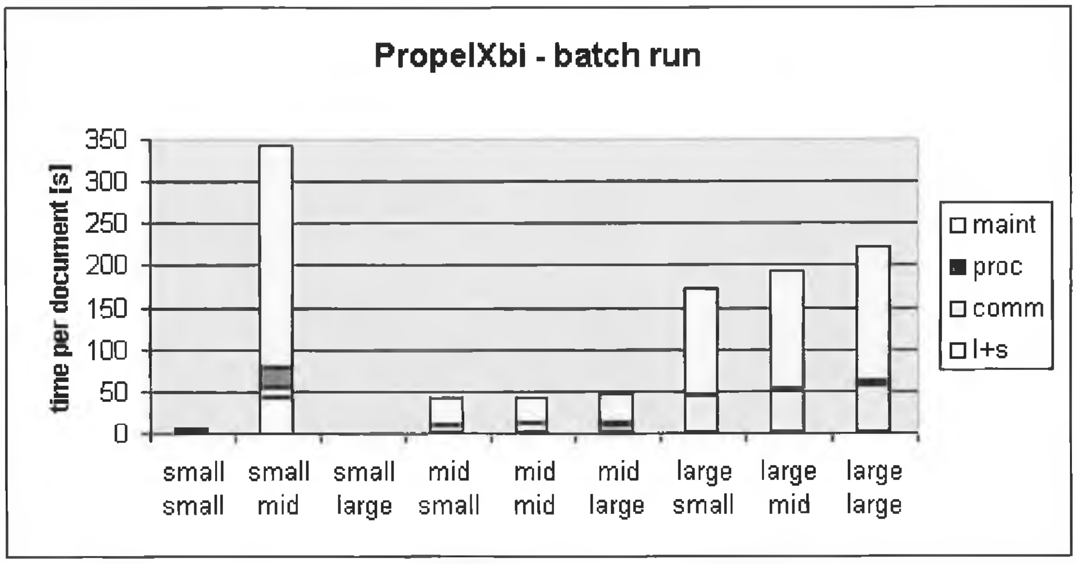
In graphs Fig. 9.12 and Fig. 9.13, the Pre-processing and Post-processing stages were bundled into the Maintenance category, Core-processing is called by the simple term Processing and the time spent in passing documents from one component to another is in the Communication category. A special category is given to time spent by loading documents in the pipe and saving them to the disk. Each column represents the result for given pipeline and a sample document run through it.

The first thing, which can be seen from the acquired results is the immense amount of time spent by PropelXbi on doing maintenance work, which is accounting for around 80 % of all the processing time. This is due to the different architectures of PropelXbi and compiled pipelines. In the compiled pipeline, when a document is handed to an XComponent to process, the transforming Java class is loaded into memory or simply invoked if it was loaded to memory already. In PropelXbi, though, the receiving Message Driven Bean uses RMI to call another EJB called the Executive, and requests the appropriate XComponent. The Executive then passes back the transformation code and the MDB executes it. As another maintenance operation, the MDB extracts the document from the received JMS message before the execution of the XComponent and then, after the execution, it creates a new JMS message and incorporates the transformed document in it. All these three processes prolong the Pre-processing and Post-processing stages. In contrast, in the compiled pipeline, documents are passed directly and no additional document-related processing needs to take place. This results in most of the processing time being spent on actual useful transformation, which is demonstrated by the high ratio of time spent in Core-processing by the compiled pipelines.

The second aspect, which can be observed in our performance data, is the efficiency of the JMS communication system. In PropelXbi, documents are passed between XComponents by wrapping them in a JMS message and placing then in a processing queue, from where they are picked up by the MDB, which executes the code of the successive component. In the case of compiled pipelines, documents are either kept in memory and passed to the following components directly, or if transforming code doesn't allow saving its output to a memory a temporal file is used to save intermediate results. As shown in the graphs, communication in both devices contributes with less than 1 % to the overall processing time and thus proves to be very efficient.

Analysis of slow-down in batch runs

In batch runs, where multiple documents are submitted to a processing pipeline at once, PropelXbi shows to be even more inefficient than in single runs. The cause of this inefficiency is PropelXbi's architecture. When multiple documents are in PropelXbi's pipeline, multiple EJB's attempt to process inserted documents and contend for processing and memory resources. This contention leads to time delays and less effective use of allocated resources. Furthermore, as the EJB's require large amounts of memory, the more EJBs that are running, the less they are efficient, as they quickly use all available memory. In such case, an EJB has to use only limited amount of memory, which is allocated to it and performs less efficient than it would if it had the ideal quantity of memory. In compiled pipelines, there isn't any corresponding problem with multiple documents, as they were designed to process only one document at a time and don't use EJB technology. Following figure depicts the mentioned aspect of EJB contention.



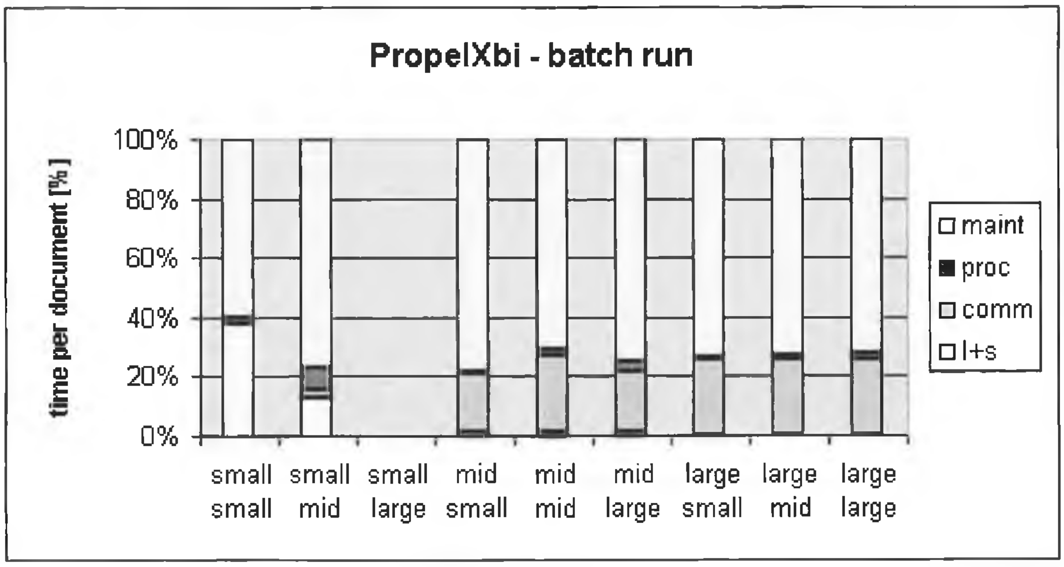


Fig. 9.14 Proportion of PropelXbi transformation stages in batch run

Fig. 9.14 clearly shows that in the case of large pipes, the communication stage grew significantly. In PropelXbi, the number of EJB's is fixed and thus when there are more documents than EJB's, these documents which can't be processed at the moment are placed in a queue, where they wait for next EJB which becomes free. This idle waiting causes growth of the communication stage. In the case of a small pipeline, the communication delay didn't play an important role, as PropelXbi managed to process submitted documents before the next ones were submitted and thus resource contention didn't occur. As processed documents in this pipeline were of large size, the saving and loading stage became longer than in the other cases.

The exceptional length of the processing time of the mid-size document in a small pipeline (5 minutes 42 seconds) led to an interesting observation. It seems that when we increase the size of documents, PropelXbi at some stage reaches a point of congestion, after which its performance declines with significantly higher rate. To test this behaviour, we used 5, 10 and 15 stage pipelines based on the small pipeline, used in first test. Through these pipes, we run batches of 20 files of sizes 50KB, 100KB, 150KB … to 600KB. The following figure shows the average processing times pre document we obtained.

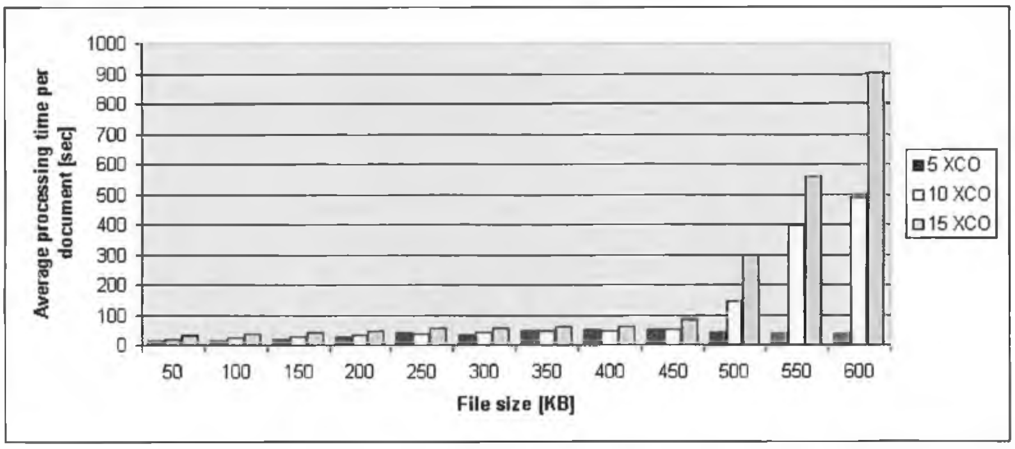


Fig. 9.15 Change of average processing times with respect to file size in batch runs

Note: Maximal available memory used in these tests was 512 MB.

Fig. 9.15 shows that the congestion point for longer pipelines lies between 450 KB and 500 KB and from the first test we know, that the congestion of the small pipe is located between 600 KB and 800 KB. PropelXbi gets to a congestion point when the majority of available memory is used for EJB's and documents in being transformed. In such a situation, processing suffers by lack of available memory and competing for scarce memory resources causes overhead, which hinders performance in a significant scale. The obtained results show an interesting fact that the location of this congestion point doesn't depend on the size of pipeline, but on the size of the files being processed and maximum amount of the available memory.

The independency on the pipeline size is a result of PropelXbi's architecture. There is a fixed number of EJB's, which can be allocated and even if the number XComponents in a pipeline is greater, only that number of EJB's is loaded to memory. As number of EJB's is fixed, the only other thing which can consume available memory is the data being processed. Because of that, the location of congestion point depends on the size and number of files being in PropelXbi at the same time and amount of memory available. In shorter pipelines, transforming the engine manages to process the documents faster and so there are fewer documents in the pipeline at one time and thus less memory is used. This results in shifting the congestion point to the higher sizes as in the case of pipeline consisting of 5 components. With a higher limit of maximum available memory, the congestion point would shift to greater file sizes as well.

9.3.3 Conclusion and Improvement Suggestions

The performed tests showed that the performance of compiled pipelines is superior to the performance of current PropelXbi transformation engine. In single file runs, the compiled pipelines performed 3 to 5 times faster, occasionally even 10 times faster. In batch runs, where multiple documents were processed at the same time, the performance difference was even greater with the compiled pipelines being 3 to 6 times faster, in one case reaching an exceptional 35 times faster execution.

The cause of the inefficiency of PropelXbi transformation engine was identified to be its architecture built on Enterprise Java Beans. It shows that use of EJB's is counterproductive as their maintenance takes an average of 19 times more time (in a single run) than the execution of the actual document transformation. In batch runs, the maintenance cost rises to an even worse ration of an average of 57 times the time of processing. As can be expected, in a case of small document sizes, ratio of time spend by maintenance and the time of actual transformation is considerably higher, as the transformation time is small compared to the maintenance time which stays rather stable irrespective of the document size.

The architecture based on EJB's was developed with objective that Message Driven EJB's (MDB's) would automatically allocate themselves to parts of the pipeline, where their work is most needed. Furthermore, if the MDB pool was clustered over more servers, the work could be physically distributed over more machines and thanks to that, processed in parallel. Unfortunately, it shows that the overhead associated with their maintenance is too big compared with the transformation work they are supposed to do, and impedes all potential advantages they could bring. It is possible that EJB architecture would bring some benefit if the MDB pool was spread over several machines, but performed tests suggest, that the maintenance overhead would override this performance gain as well.

As performance is not the only criterion when examining the quality of a system, we look on other features of PropelXbi and XCompiler as well. The following table overviews features and qualities of PropelXbi and the compiled pipelines execution runtime.

Features	XCompiler	PropelXbi
Java technology	J2SE	J2EE
Communication system	Direct calls	JMS, RMI
Speed	High	Low
Required memory	Small	Large
Stable performance	Yes	No
Monitoring	---*	Yes
Exception notification	---*	Yes
Simultaneous processing	---	Yes
Innate distributability	---	Yes

Tab. 9.3 Feature comparison of XCompiler and PropelXbi

Note: * this feature can be easily implemented.

The table first records our finding that compiled pipelines achieve superior speed of execution compared to PropelXbi. As another recorded feature, the memory footprint and requirements are observed. The execution of compiled pipelines requires only plain JVM, setting small a memory requirement, whereas PropelXbi needs the whole J2EE application server running for its functioning. As the XCompiler transforms one document at a time, with increased number and size of submitted documents, its processing time increases linearly. In contrast, as shown in previous analysis, in PropelXbi, increasing the number and size of submitted documents does not result in linearly increased processing time. After a certain point is reached, the execution time of transformations in PropelXbi increases super-linearly.

The current compiled pipeline execution runtime does not provide monitoring events and exception notification in a way, which could be used by other applications. However, it does provide this information by printing it on the screen. As such information is available, if there was a need to provide these features to other applications, they could be easily implemented by providing event notifications in addition to current visual presentation. We already mentioned several times, that the XCompiler was not designed to allow processing of multiple documents at the same time and thus it does not have the feature of simultaneous processing. Even though, PropelXbi can process several documents at the same time, it showed that simultaneous processing decreases its performance and doesn’t bring any advantage. The last mentioned feature is intrinsic distributability. J2EE architecture was designed with vision of distributed EJB pools in mind and thus PropelXbi innately provides this feature. In case of XCompiler, distributed processing can be implemented with help of Grid-technologies, which implementation is topic of next chapter 10.

Improvement suggestions

Our suggestion is to keep the current infrastructure for the development of XPipes and XComponents and instead of the current EJB-based transformation engine, use a different one, based on plain Java class invocation (for example compiled pipelines execution runtime). In that way, the advantages of both are attained. The current XPipe development environment would provide easy construction and maintainability of pipelines and a plain Java transformation engine would provide good transformation speed.

There are several possibilities to acquire such a transformation engine. One option is to compile the pipelines before execution using XCompiler and use a script or a direct Java call to start compiled pipeline execution runtime. In the current state, execution runtime provided by XCompiler doesn't provide three facets, which are present in the current EJB solution. They are: monitoring information, error and exception handling and the possibility to submit multiple documents at the same time. More precisely, monitoring and error handling is present in the current compiled pipe runtime, as all the necessary monitoring and exception information is written to screen, but can't be reasonably used by other Java applications. This can easily be changed to allow such usability, though. The compiled pipeline runtime was not designed to process multiple documents at the same and thus doesn't allow it. If such feature was needed, an extended version could be written, which would create multiple instances of the runtime, while examining available memory at the same time, so that exceptions caused by lack of memory resources wouldn't occur. Our discussion of potential modifications of current XCompiler code has led us to a second option, which is taking the current XCompiler as a base and extending it so that it provides additional features and/or better integrates with other code used in PropelXbi.

The other option would be to write a new engine, which would use the same concept as XCompiler, which is a simple Java class invocation. In that case, documents could be passed between individual XComponents in the same way as in the XCompiler, which is by saving them in memory and passing them directly to successive components without the need of writing them to disk. Optional writing to disk could be used as well though, as a security matter against unexpected crashes of execution system. As such an engine would be integral part of PropelXbi, it would not need to observe the limitations set on XCompiler, like for instance that all generated code needs to be Java class or jar. Without that limitation, Jython and an XSLT code could be kept in its original form and appropriate processors could be used to execute them, which would result in more effective execution.

If there was a strong need to keep the current EJB architecture, in spite of its vast inefficiency, modification in document submission would be another option. Tests have shown that multiple documents present in the execution engine at one time cause its slow-down and thus a throttling device would improve its performance. A throttle would be placed in front of PropelXbi's input and would hold submitted documents, so that there is only one document being processed at one time. It seems counterintuitive to delay work, which needs to be done, but tests have shown that delay caused by concurrent processing of multiple documents is many times longer than processing time itself. Because of that, letting only one document in, at a time would result in a shorter overall transformation time of the whole batch of documents compared to the situation if they were submitted all at once.

Chapter 10

Distributed XML processing

10 Distributed XML processing

To examine how the concept of parallel processing can be utilised for PropelXbi and XML document processing, we implemented a distributed version of the document processing system. First, in 10.1 we present the distributed processing system we assembled. Then, in 10.2 we list a set of questions designed to evaluate the usefulness of the parallel processing system used. Then, in 10.3 we draw theoretical solutions to our questions and in 10.4 we show the results we got from real measured data and discuss the differences with the theoretical solutions. In enclosing section 10.5 we provide a conclusion about efficacy of the use of parallel processing for document transformations.

10.1 Distributed processing system

We've decided to use compiled pipelines, created by the XCompiler from the previous chapter, and assembled a distributed system which allows simultaneous use of multiple computers for transformation of a large number of documents. As there are many already existing packages which take care of work management in distributed environments (as reviewed in chapter 7.3), we decided to use one of them. Namely we chose to use publicly available package Condor 6.6.2. The structure of the implemented processing system is shown on Fig. 9.16

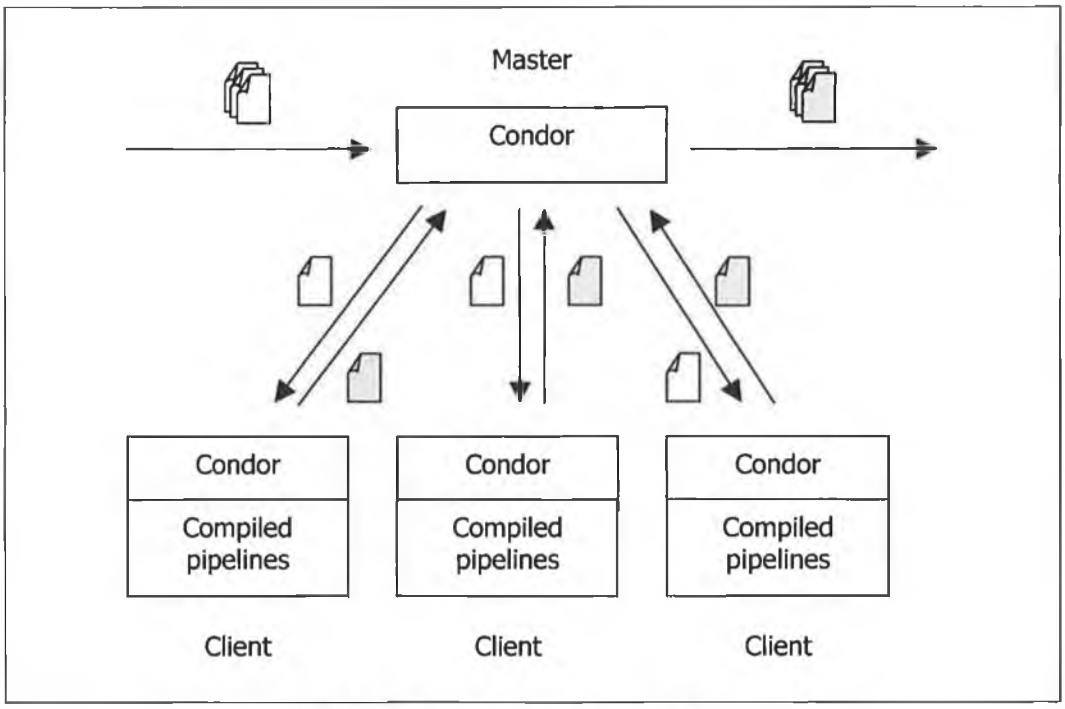


Fig. 9.16 Distributed processing system

One machine was selected as a Master and was used for document submission and management of work distribution to client machines. The Condor service on client machines received documents from the Master and submitted them to the appropriate compiled pipelines according to additional instructions received from the Master. When the processing of a document finished on a client machine, the resulting document was sent to the Master and client waited for further documents to process.

Our distributed processing system consisted of one Master machine with Condor and the submitting scripts and twenty Client machines with Condor and identical file structures of compiled pipelines. All the machines had Pentium 4 2.2GHz processors, 256 MB RAM and all were connected through a local network. We could have used PropelXbi as a transformation device on client machines as well. The only difference in context of this chapter is that compiled pipelines perform the document transformation faster.

10.2 Questions we were asking

By assembling distributed processing system, we decided to investigate three important questions, which arise when contemplating utilisation of distributed processing.

1) What is the efficiency of distributed processing? Given the time of serial processing, what would be the processing time with different numbers of processors?

2) Is there a limit after which adding more processors is not beneficial? Can we identify that limit?

3) Given a workload and a desired processing time, how many processors would we need to achieve desired processing time?

10.3 Theoretical solution

Parallel processing has been studied since the nineteen-sixties and thus we can base our analysis on previous findings.

10.3.1 Parallel processing time

Concerning our first question about efficiency of parallel processing, Gupta and Kumar (Gupta & Kumar 1993) give following formula stating what is the parallel processing time, with relation to the number of processors and time of the sequential processing.

$$T_P = \frac{W + T_O(W,p)}{p} \qquad (10.1)$$

T_P is the parallel execution time, i.e. time from the start of a parallel computation to the moment the last processor finishes execution. W is the problem size measured as the number of operations needed to solve the problem. The serial execution time T_S (time to process given problem) is determined as $T_S = t_c W$ where t_c is a machine dependent constant. p is the number of processors and $T_O(W,p)$ is the total parallel overhead. T_O is the sum total of all overhead incurred due to parallel processing by all processors and can be expressed as $T_O = pT_P - T_S$. The total parallel overhead is a function of problem size and the number of processors.

For our case, we change (10.1) so that it more naturally reflects the components of total processing time.

$$\begin{gathered} T_P = \frac{nT_S}{p} + \frac{T_O(W,p)}{p} \\ T_P = \frac{n}{p}T_S + h(n,p) \end{gathered} \qquad (10.2)$$

In our context, the problem size is equivalent to a number of processed documents n multiplied by an average sequential processing time T_S (the average time needed to process one document). The total parallel overhead divided by the number of processors is replaced by the overhead function $h(n,p)$. The first term on the right hand side represents ideal processing time and the second term represents the additional parallel processing overhead. In our study we are interested in predicting parallel processing time for a group of similar documents of the same size and thus we will not consider the overhead function to be dependent on the size of the document. As the size of the document would be given it would be reflected in overhead function only as additional constant.

To be perfectly exact, we infer that the first fraction of the equation should be ceiled as it expresses how much time is used by useful document processing of n documents on p processors. The ceiling function of x (noted as $\lceil x \rceil$) is the smallest integer greater than or equal to x. It's easy to see that if we have for example 6 processors and 10 documents, the total time spent by document processing would be $\lceil 6/10 \rceil T_S = 2T_S$ and not $6/10\, T_S = 1.2T_S$. The equation given by

Gupta and Kumar is meant for systems with a high number of processors and large problem sizes, in which case such a difference is not as significant. In our case though, where number of processors is likely to be small, we shouldn't discount it and final form of formula for parallel execution then is:

$$T_P = \left\lceil \frac{n}{p} \right\rceil T_S + h(n,p) \qquad (10.3)$$

The overhead function is unique to each parallel system (i.e. parallel algorithm and parallel architecture). Parallel execution time for various numbers of processors can be predicted once the form of overhead function is determined.

For example, if we assume that the overhead is linearly dependent on the number of processors and the number of documents, we can express the overhead function as:

$$h(n,p) = c_n n + c_p p + c_c \qquad (10.4)$$

where c_n, c_p and c_c are constants. For such an overhead function with coefficients $c_n = 2.5, c_p = 2, c_c = 10$ and average serial processing time $T_S = 10$, the parallel processing time can be graphed as following:

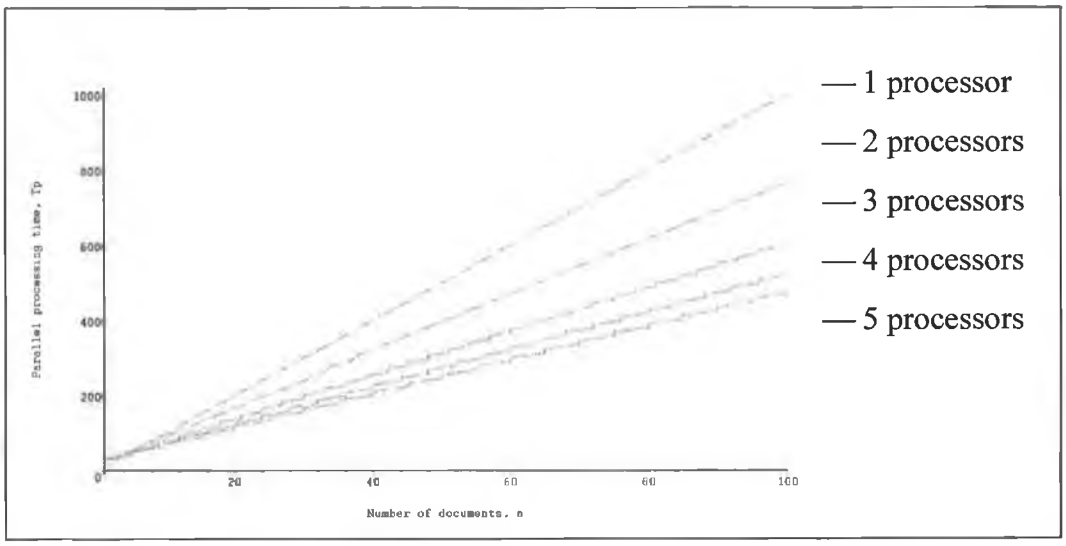


Fig. 9.17 Theoretical parallel processing time with respect to number of documents

So, if it took $10s$ to process one document sequentially and the overhead function of our parallel system was $h(n,p)=2.5n+2p+10$, then the processing of 100 documents on 3 processors would take $T_p = \left\lceil \frac{100}{3} \right\rceil 10+2.5*100+2*3+10=606s$ compared to $1000s$ which it would take if the same number of documents were processed sequentially.

Fig. 9.17 shows that the speedup we gain, decreases with increasing number of processors. This is in accord with common behaviour of parallel systems, described for example in (Gustafson 1988; Gupta & Kumar 1993). The consequence of this decrease is investigated by our next question.

10.3.2 Maximal number of beneficial processors

As the number of processors increases, the parallel processing time for fixed number of documents reduces, but with decreasing speed. At the same time the overhead increases as shown in (Gupta & Kumar 1993)[2]. At some point, the parallel processing time starts increasing again as overhead grows faster than the additional computation power offered by increasing the number of processors. After this point, it is not beneficial to add more processors. This behaviour is shown on the following graph, depicting the decrease of parallel processing time with respect to the number of processors. The equation coefficients used for figure Fig. 9.18 are the same as those used in Fig. 9.17.

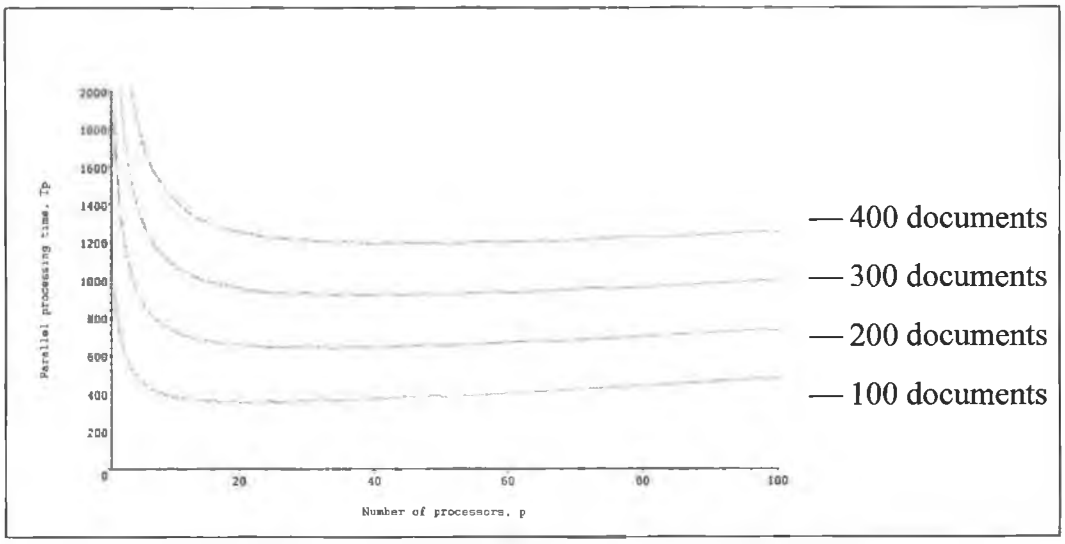


Fig. 9.18 Theoretical parallel processing time with respect to number of processors

When using the maximal number of beneficial processors, $p_{\max}$, the parallel processing time is at its lowest value and therefore $p_{\max}$ can be obtained as a solution of the differential equation

$$\frac{\partial T_P(n,p)}{\partial p} = 0 \qquad (10.5)$$

[2] Gupta and Kumar state that such point exist for parallel systems where $T_O > \Theta(p)$. Overhead function $h(n,p)$ equals to T_O / p and thus such point exists for systems with h of at least linear order.

Unfortunately, derivation of the ceiling function in the first term of the expression for T_p can't be expressed analytically. We have to reach for an approximation by omitting the ceiling function from the expression. After such modification, we can express the maximum number of beneficial processors as

$$\tilde{p}_{\max} = \sqrt{\frac{nT_s}{\frac{\partial}{\partial p} h(n,p)}} \qquad (10.6)$$

When the overhead function is linear and of the form $h(n,p) = c_n n + c_p p + c_c$, expression (10.6) simplifies to

$$\tilde{p}_{\max} = \sqrt{\frac{nT_S}{c_p}} \qquad (10.7)$$

As this value of $\tilde{p}_{\max}$ is valid for approximated T_P, the values for p in vicinity of $\tilde{p}_{\max}$ should be checked to see which value of p is the right value for non-approximated function of the parallel processing time.

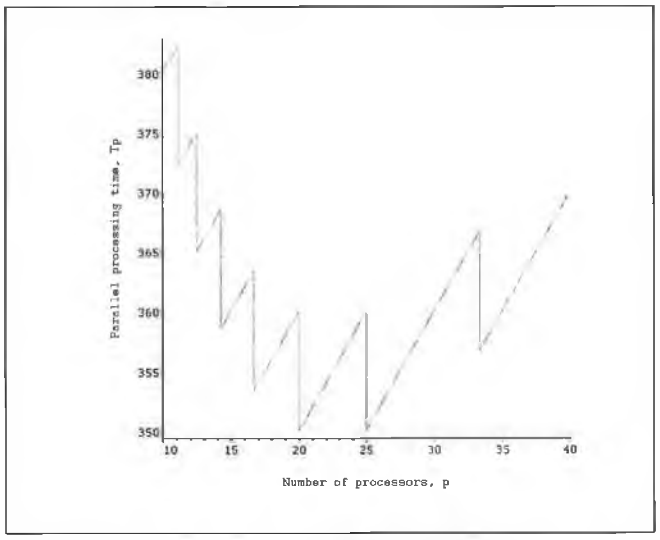


Fig. 9.19 Detail of non-approximated function of parallel processing time

If we take the parallel system from our previous example, the approximated maximal number of beneficial processors for 100 documents is $\tilde{p}_{\max} = \sqrt{100*10/2} = 22.36$. From Fig. 9.19 showing detail of the non-approximated function of parallel processing time for 100 documents, we can see that the correct $p_{\max}$ is 20.

10.3.3 Required number of processors to achieve desired execution time

The last question we were asking is: What is the necessary number of processors to achieve desired parallel processing time for given number of documents (p_{req})?

A graph, like Fig. 9.18, can be used for a first rough estimation of what is the minimal required number of processors. To get the exact value of p_{req} we have to extract it from equation (10.3). Again, as it isn't possible to extract p_{req} from the ceiling function, we have to use an approximation by omitting the ceiling of the first fraction. To be able to express p_{req} we have to know the form of the overhead function as it can also be dependent on the number of processors.

For the linear overhead function of form $h(n,p) = c_n n + c_p p + c_c$, where c_n, c_p and c_c are constants, we can extract p_{req} from (10.3) as

$$p_{req} = \left\lceil \frac{T_P - c_n n - c_c - \sqrt{D}}{2c_p} \right\rceil \qquad (10.8)$$

where D determines whether the desired execution time is achievable or not. If D is non-negative, then the given execution time can be achieved, otherwise it cannot. D is given by the equation:

$$D = T_P^{\ 2} + \left(-2c_c - 2c_n n\right) T_P + c_n^{\ 2} n^2 + 2c_n n c_c - 4c_p n T_S + c_c^{\ 2}$$

(10.9)

To give an example, we take the parallel system from the previous illustration and ask how many processors we would need to have 100 documents processed in a time of $500s$. D equals 49600, which is non-negative and thus it signifies that such a time can be reached. Using the calculated D, we get $p_{req} = \left\lceil (500 - 2.5 * 100 - 10 - \sqrt{49600})/(2*2) \right\rceil = \lceil 4.32 \rceil = 5$. This means that we need at least 5 processors to have 100 documents processed in a time of at most 500 seconds. The exact value of the parallel processing time of our system for 100 documents and 5 processors is 470 seconds. For 4 processors, it is 518 seconds. This demonstrates that formula we have devised is correct.

To have an idea of how many processors we would need for various combinations of the desired time and number of documents, we can graph p_{req} with respect to the desired time.

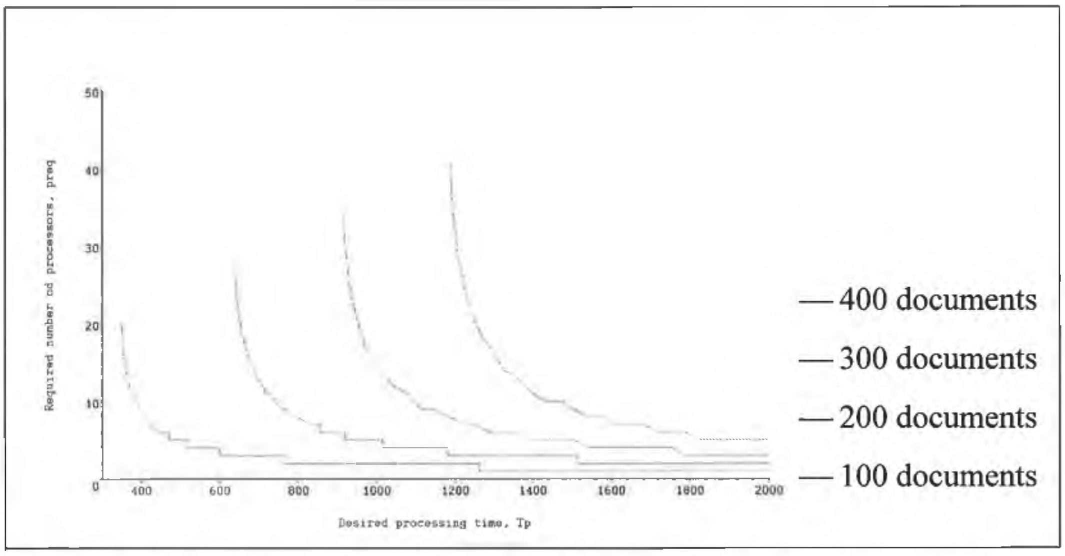


Fig. 9.20 Required number of processors with respect to desired processing time

Fig. 9.20 shows how many processors we would need to process the given number of documents in a time less than or equal to the given processing time.

It shows that with a shorter desired processing time, the number of required processors grows significantly.

10.4 Actual performance results

To verify validity of formulas devised in previous section, we carried out extensive performance tests on system described in 10.1. In our test, we measured execution times of batches of documents sent to be executed on different numbers of processors.

The following figure shows the measured parallel processing times.

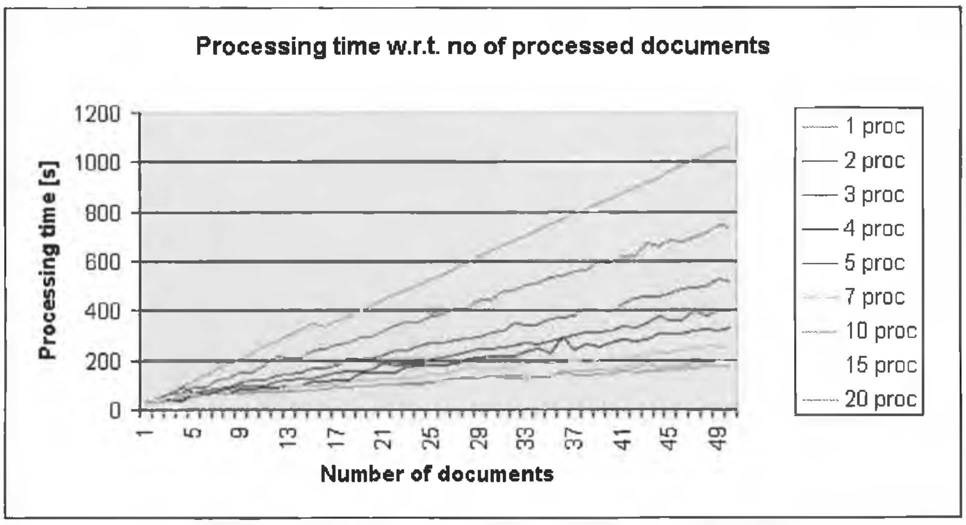


Fig. 9.21 The measured parallel processing times with respect to the number of documents

Note: Large size version of Fig. 9.21 can be found in Appendix C as Fig. C.1

The highest line in Fig. 9.21 shows the processing time for the sequential processing of the given number of documents. The other lines are the processing times of executions when different numbers of processing machines are available. It shows that in accord with the predicted performance, the speed of decrease of processing time lessens with higher number of processors. E.g. the difference between the processing on 15 and 20 machines is negligible compared to difference between 1 and 2 processors. In contrast with the theoretical prediction, real processing times don't exhibit "steps" which are caused by the ceiling function in the theoretical run.

The next figure shows how the measured processing times relate to the number of processors on which the documents were processed.

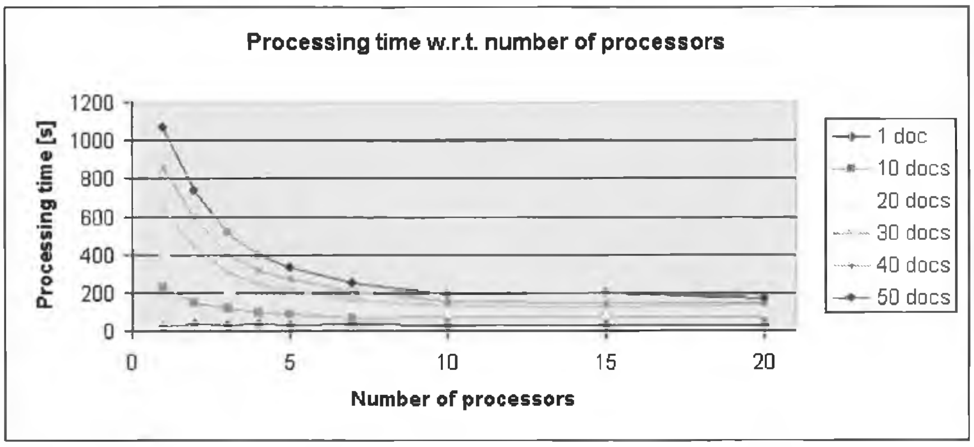


Fig. 9.22 Measured parallel processing time with respect to number of processors

Note: Large size version of Fig. 9.22 can be found in Appendix C as Fig. C.2

Fig. 9.22 shows, again, that the measured data confirm the devised formulas for the times of parallel processing. In this graph, the decrease of gain that we get from adding more processors is even more visible than in Fig. 9.21. The data we got doesn't show any performance knee, that is, when the processing time starts to grow again. We did not get to the high number of processors necessary for this effect. Nevertheless, in real time scenarios, we don't expect the number of available processors to be considerably higher in any case.

Important characteristic of parallel systems are the overhead functions. The overhead in the system we measured is depicted in the next two graphs.

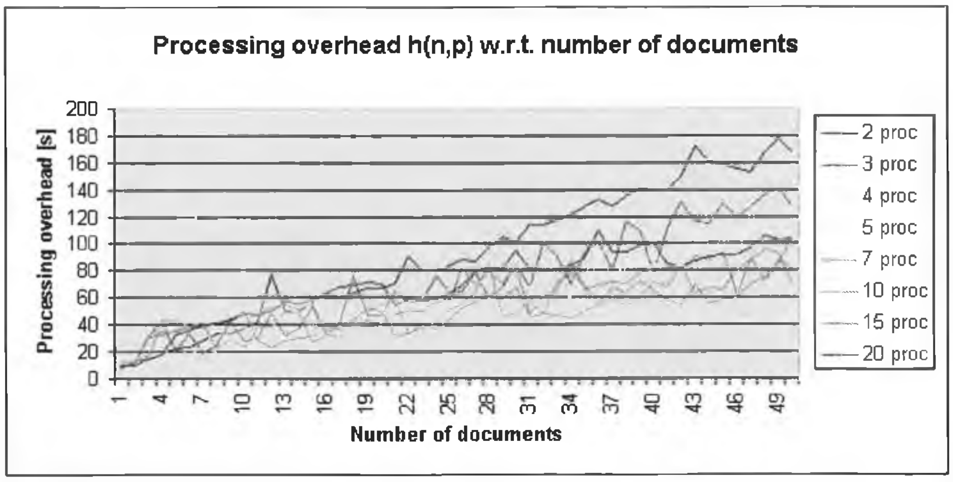


Fig. 9.23 Processing overhead with respect to number of documents

Note: Large size version of Fig. 9.23 can be found in Appendix C as Fig. C.3

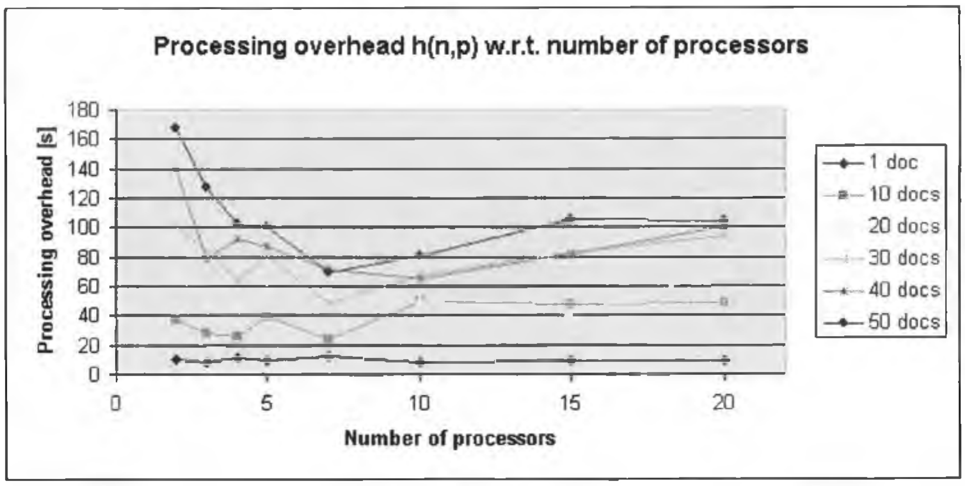


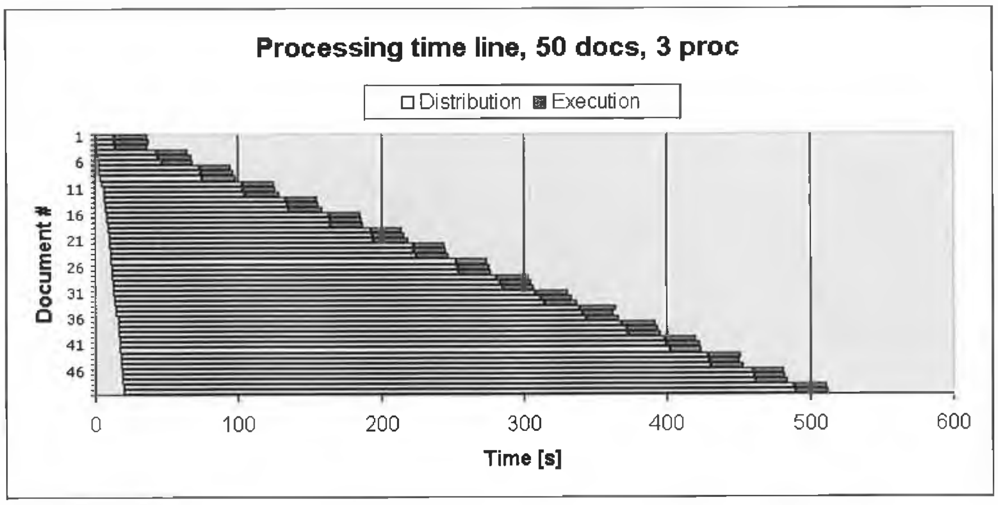
Fig. 9.24 Processing overhead with respect to number of processors

Note: Large size version of Fig. 9.24 can be found in Appendix C as Fig. C.4

The displayed overhead was calculated as the difference between the measured parallel processing time and the ideal time, i.e. the processing time of the parallel system without any additional overhead.

As predicted, it shows that the overhead function is dependent on the number of processors as well as on the number of documents. The measured runs of the overhead exhibit uneven growth and have some noise superimposed. We assume that these irregularities were caused by stochastic behaviour of the interconnecting LAN and Condor work distribution mechanism.

In general, the overhead grows with the increasing number of documents and lessens with the increasing number of processors. However, Fig. 9.24 shows that the overhead decreases until the number of processors is seven, after which it is higher again. We ascribe this behaviour to Condor's inability to utilise the higher number of available machines as efficiently as smaller numbers. This can be seen from the detailed views of individual runs, as shown in the following figures.



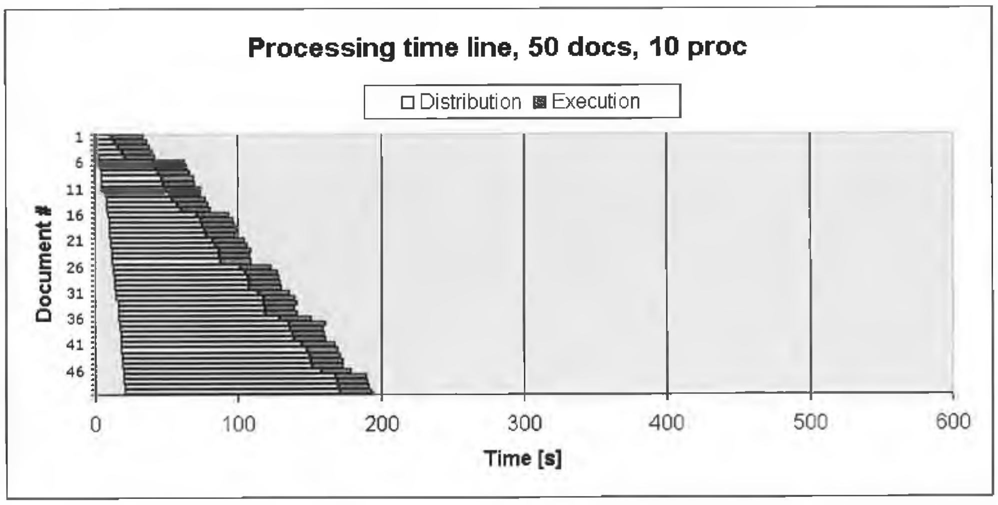


Fig. 9.25 Processing timelines for run on 3 and 10 processors

The timelines show the development of the processing of 50 documents on a system with three and a system with ten available machines. Each line shows the processing of one document. It starts when the document was submitted for processing, continues, with the next part denoting how long it took before the document started to be processed and the final section of the line displays the length of the actual document transformation. It can be seen that, in a system with three processors, Condor managed to distribute the documents so that their processing happened fairly simultaneously. On the contrary, in case of a ten processors system, Condor didn't achieve simultaneous execution on all the available machines, and thus in incurred higher idle time and overhead.

10.5 Conclusion

To test how the concept of distributed computing is utilisable for PropelXbi, we have implemented a Grid-based distributed version of PropelXbi. Our tests have shown that it is possible to utilise the concept of distributed computing for increasing the performance of document transformation by PropelXbi. As expected, the distributed document processing system behaves in the same way as other parallel processing systems. An important feature of such behaviour is the decreasing gain which we get from adding more processors and thus there is a maximum number of beneficial processors that actually bring you any real advantage. This number of processors, along with the theoretical prediction of the parallel processing time, should be considered when deciding whether to use a distributed system.

In this chapter, we proposed formulas which can be used for the prediction of total parallel processing time, maximal number of beneficial processors and the required number of processors to achieve the desired processing time of a given number of documents. These formulas can be used for deciding about the effectiveness of employing distributed processing for increasing the performance of a document processing system.

CHAPTER 11

CONCLUSION AND FUTURE WORK

11 Conclusion and Future Work

In this chapter, we first summarise major findings of this thesis in 11.1 and then we indicate the directions of potential future work in 11.2.

11.1 Conclusion

In this thesis, we first reviewed the relevant architectures and enhancement techniques used in parallel processing and we found that all of these are already present in an appropriate form in PropelXbi. These enhancements are pipeline processing, instruction pre-fetch, caching, data forwarding and vector pipeline chaining.

Next, we presented the Jackson Inversion, which comes with concept of compilation of processing components. It shows that such a compilation is beneficial only when the component's processing time is short. When the processing time is short then the incurred loss of parallelism is negligible.

The subsequently reviewed concept of TupleSpaces offers a way of expanding PropelXbi from a single machine to distributed computing. However, the goal of distributed computing can be reached more conveniently by using Grid technologies.

The next topic, Project JXTA, comes with an inter-machine communication mechanism which is independent of the machines' software and hardware. Yet, in PropelXbi, there already is a communication system based on JMS and its advantages outweigh those offered by JXTA.

Grid computing technologies, which were next looked at, provide a way to distribute document processing on multiple machines so that the documents can be processed in parallel. This results in increased performance of the processing system. We devised an architecture for the distributed version of PropelXbi which builds on Grid technologies and uses their advantages.

After examining the relevant techniques with potential to streamline PropelXbi's performance, we implemented two enhancements which had the highest potential to improve PropelXbi's performance. The first was an Off-line XComponent compiler and J2SE-based compact version of PropelXbi runtime (compiled pipelines), which was built on the concept of Jackson Inversion. Tests showed that the compact version of PropelXbi runtime achieves significantly better performance than the original J2EE version. The second implemented enhancement was a Grid-based distributed version of PropelXbi. Tests showed that the distributed processing can be used for streamlining PropelXbi's performance and that the distributed version follows the same laws as other standard parallel processing systems. Furthermore, it demonstrated that expansion from single machine processing to distributed computing can be conveniently achieved without the need to alter the current runtime code of PropelXbi in any way.

The importance of this work lies in the identification of significant enhancements for increasing the efficiency of PropelXbi. In addition, the identified enhancements can also be used in the design of other similar large-scale document processing systems. Our testing has demonstrated that considerable improvements in performance can be achieved by utilising J2SE technology and implementing the concepts of component compilation and distributed processing. The enhancement implementations run from 3 to 5 times faster than the current version of PropelXbi. Once or twice it actually occurred that the processing was 10 to 35 times faster.

11.2 Future Work

This work presented the implementations of two enhancements. Future work can be directed in the further development and exploration of the already implemented improvements and implementation of those which weren't implemented yet.

Concerning the subject of distributed computing, other Grid scheduling systems can be inspected (e.g. N1 Grid Engine, OpenPBS) regarding efficiency and convenience of their use. In addition, further examination may be focused on the prediction of their performance under different conditions (e.g. different number of processors, sizes of available memory, different processing speeds etc.).

As a second suggested direction of work, an off-line XComponent compiler can be further developed so that it provides supplementary features present in the current PropelXbi (e.g. monitoring capabilities, support for Scatter/Gather components etc.).

A final suggestion is to implement an on-line XCompiler, which would use online information about the current run of document processing and would compile XComponents according to it so that the overall speed of processing is increased.

REFERENCES

References

Aberdeen Group (2002), *Sun's Grid Computing Solutions Outdistance the Competition*, May 2002. Retrieved: 25 September 2003, from http://wwws.sun.com/software/gridware/sge_aberdeen.pdf.

Aloisio, G., Cafaro, M., Epicoco, I., Fiore, S. & Williams, R., *The Grid Resource Broker*. Retrieved: 25 September 2003, from http://sara.unile.it/grb/grb.html.

Aloisio, G., Cafaro, M., Epicoco, I., Fiore, S. & Williams, R., *Grid Resource Broker User Manual*. Retrieved: 25 September 2003, from http://sara.unile.it/grb/grbusermanual.doc.

Altair Grid Technologies (2003a), *OpenPBS*. Retrieved: 8 August 2003, from http://www.openpbs.org/.

Altair Grid Technologies (2003b), *PBS Pro Home*. Retrieved: 8 August 2003, from http://www.pbspro.com/.

AMWAT, *The AppLeS Master/Worker Application Template*. Retrieved: 3 October 2003, from http://grail.sdsc.edu/projects/amwat/.

Avaki Corporation (2003a), *Avaki : Home*. Retrieved: 19 March 2003, from http://www.avaki.com/.

Avaki Corporation (2003b), *Avaki Comprehensive Grid 3.0 Data Sheet*, January 2003. Retrieved: 19 March 2003, from http://www.avaki.com/global/pdf/avakicompgrid30data.html.

BEA (2002a), *Distributed Computing with BEA WebLogic Server*, 15 September 2002. Retrieved: 7 April 2003, from http://www.bea.com/content/news_events/white_papers/BEA_WL_Server_DistributedComputing_wp.pdf.

BEA (2002b), 'Data Integration', in *Introducing WebLogic Integration*, June 2002. Retrieved: 7 April 2003, from http://edocs.bea.com/wli/docs70/overview/dataint.htm.

BEA (2003a), *Achieving Scalability and High Availability for E-Business*, 27 March 2003. Retrieved: 7 April 2003, from http://www.bea.com/content/news_events/white_papers/BEA_WL_Server_Clustering_wp.pdf.

BEA (2003b), 'WebLogic Server Services', in *Introduction to WebLogic Server and WebLogic Express*, 12 February 2003, pp. 2-4 - 2-8. Retrieved: 7 April 2003, from http://edocs.bea.com/wls/docs81/pdf/intro.pdf.

BEA (2003c), 'Load Balancing in a Cluster', in *Using WebLogic Server Clusters*, 27 March 2003, pp. 4-5 - 4-16. Retrieved: 7 April 2003, from http://edocs.bea.com/wls/docs81/pdf/cluster.pdf.

Bent, J. & Thain, D. (2002), *Condor Tutorial*, July 2002. Retrieved: 31 March 2003, from http://www.cs.wisc.edu/condor/tutorials/condor-hpdc11.ppt.

Borgstorm, R. S. (2000), *A Cost-Benefit Approach to Resource Allocation in Scalable Metacomputers*, Doctor of Philosophy, The Johns Hopkins University. Retrieved: 18 August 2003, from http://www.cnds.jhu.edu/pub/papers/ryan-thesis.pdf.

Bosak, J. (1996), 'DSSSL Online in context', in *DSSSL Online Application Profile*, 16 August 1996. Retrieved: 13 September 2004, from http://www.ibiblio.org/pub/sun-info/standards/dsssl/dssslo/do960816.htm.

Capello, P. (2003a), *JANET*. Retrieved: 5 September 2003, from http://www.cs.ucsb.edu/projects/janet/.

Capello, P. (2003b), 'Janet's Abstract Distributed Service Component', in *Proceedings of the 15th IASTED International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, California, USA, pp. 751 - 756. Retrieved: 5 November 2003, from http://www.cs.ucsb.edu/~cappello/papers/03pcds.pdf.

Carriero, N. & Gelernter, D. (1989), 'Linda in context', *Communications of the ACM,* vol. 32, no. 4, pp. 444-458. Retrieved: 10 February 2003, from http://www.cs.cornell.edu/Courses/cs614/2003SP/papers/CG89.pdf.

Chapman, B. M., Sundaram, B. & Thyagaraja, K. K. (2002), *EZ-Grid: Integrated Resource Brokerage Services for Computational Grids*. Retrieved: 20 August 2003, from http://www.cs.uh.edu/~ezgrid/EZ-GridAbstract.pdf.

CNDS, *The Frugal System*. Retrieved: 18 August 2003, from http://www.cnds.jhu.edu/research/metacomputing/frugal/.

Cocoon (2003), *The Apache Cocoon Project*. Retrieved: 28 January 2003, from http://cocoon.apache.org/.

Collab.Net (2003a), *General JXTA FAQ*. Retrieved: 25 February 2003, from http://www.jxta.org/project/www/docs/DomainFAQ.html.

Collab.Net (2003b), *Project JXTA Homepage*. Retrieved: 25 February 2003, from http://www.jxta.org/.

Collab.Net (2003c), *Project jxtaSpaces*. Retrieved: 11 February 2003, from http://jxtaspaces.jxta.org/.

Commerce One, *xCBL-index*. Retrieved: 20 April 2004, from http://www.xcbl.org/.

Condor, *The Condor Project Homepage*. Retrieved: 25 March 2003, from http://www.cs.wisc.edu/condor/.

Condor Team (2003a), *Condor Version 6.4.7 Manual*, 7 February 2003. Retrieved: 25 March 2003, from http://www.cs.wisc.edu/condor/manual/v6.4/condor-V6_4-Manual.pdf.

Condor Team (2003b), 'Frequently Asked Questions (FAQ)', in *Condor Version 6.4.7 Manual*, 7 February 2003. Retrieved: 25 March 2003, from http://www.cs.wisc.edu/condor/manual/v6.4/7_Frequently_Asked.html.

Condor-G, *Condor versus Condor-G - what's the difference?* Retrieved: 25 March 2003, from http://www.cs.wisc.edu/condor/condorg/versusG.html.

Contivo (2001), *Contivo product brochure*, 2001. Retrieved: 07 February 2003, from http://www.contivo.com/about/brochure.pdf.

CoverPages-TS (2002), *The Cover Pages website. Technology Reports: Tuple Spaces and XML Spaces*, (Last update: 11 October 2002). Retrieved: 10 February 2003, from http://xml.coverpages.org/tupleSpaces.html.

CSEP (1995), *Computational Science Education Project. Computer Architecture*, 1995. Retrieved: 12 November 2002, from http://www.phy.ornl.gov/csep/ca/ca.html.

cXML, *cXML homepage*. Retrieved: 15 May 2004, from http://www.cxml.org/.

D.I.B. (2002), *e-Business Enabler D.I.B!!* Retrieved: 28 January 2003, from http://www.dib.net/eng/pro/pro_xml_04.asp.

Dail, H., *UCSD GrADS Page*. Retrieved: 3 October 2003, from http://gcl.ucsd.edu/~grads/.

DataConcert (2003), *DataConcert | Technology*. Retrieved: 28 January 2003, from http://www.dataconcert.com/technology.asp.

Dongarra, J. (2003), *CS 594 Spring 2003 Lecture 3: Overview of High-Performance Computing*. Retrieved: 20 April 2004, from http://www.cs.utk.edu/~dongarra/WEB-PAGES/SPRING-2003/lect03.pdf.

Duncan, R. (1990), 'A survey of parallel computer architectures', *IEEE Computer*, vol. 23, no. 2, pp. 5-16. Retrieved: 20 April 2004, from http://csdl.computer.org/dl/mags/co/1990/02/r2005.pdf.

ebXML, *ebXML - Enabling A Global Electronic Market*. Retrieved: 11 April 2003, from http://www.ebxml.org/.

EZ-Grid, *EZ-Grid - Introduction*. Retrieved: 20 August 2003, from http://www.cs.uh.edu/~ezgrid/.

FIX Protocol (2004), *The FIX protocol Organization*. Retrieved: 20 April 2004, from http://www.fixprotocol.org/.

Foster, I. (2002), *The Challenges of Grid Computing*, 4 March 2002. Retrieved: 25 March 2003, from http://www.cs.wisc.edu/condor/presentations/PC-2002/foster.ppt.

Foster, I. & Kesselman, C. (eds.) (1999), *The GRID: Blueprint for a New Computing Infrastructure*, Morgan Kaufman Publishers, Inc., San Fracisco, California, USA.

Foster, I. & Kesselman, C. (2001), *Grid Computing.* Retrieved: 24 March 2003, from http://wwwinfo.cern.ch/seminars/2001/2001-OtherFormats/t-010117.pdf.

Foster, I., Kesselman, C., Nick, J. M. & Tuecke, S. (2002a), 'Grid Services for Distributed System Integration', *Computer,* vol. 35, no. 6, pp. 37-46. Retrieved: 24 March 2003, from http://www.globus.org/research/papers/ieee-cs-2.pdf.

Foster, I., Kesselman, C., Nick, J. M. & Tuecke, S. (2002b), *The Physiology of the Grid*, 22 June 2002. Retrieved: 20 February 2003, from http://www.globus.org/research/papers/ogsa.pdf.

Foster, I., Kesselman, C. & Tuecke, S. (2001), 'The Anatomy of the Grid', *International Journal of Supercomputer Applications,* vol. 15. http://www.globus.org/research/papers/anatomy.pdf.

Fox, G. C., Williams, R. D. & Messina, P. C. (eds.) (1994), *Parallel Computing Works!*, Morgan Kaufman Publishers, Inc., San Francisco, California, USA.

Fox, J. (2003), *Semantic Information Management (SIM)*, 28 October 2003. Retrieved: 20 April 2004, from http://www.simc-inc.org/archive0304/metadata/presentations/fox/fox_frame.htm.

Frey, J. (2002), *Condor-G: An Update.* Retrieved: 25 March 2003, from http://www.cs.wisc.edu/condor/presentations/PC-2002/jfrey.ppt.

GigaSpaces Technologies (2002a), *GigaSpaces Cluster*, March 2002. Retrieved: 10 February 2003, from http://www.gigaspaces.com/download/GSClusterWhitePaper.pdf.

GigaSpaces Technologies (2002b), *GigaSpaces Platform*, 25 February 2002. Retrieved: 10 February 2003, from http://www.gigaspaces.com/download/GigaSpacesWhitePaper.pdf.

GigaSpaces Technologies (2002c), *GigaSpaces Platform Product Overview*. Retrieved: 10 February 2003, from http://www.gigaspaces.com/download/GigaSpacesPlatformProductOverview.pdf.

Globus Project (2001), *Introduction to Grid Computing and the Globus Toolkit*, 12 October 2001. Retrieved: 5 March 2003, from http://www.globus.org/training/grids-and-globus-toolkit/IntroToGridsAndGlobusToolkit.ppt.

Globus Project (2002), *Globus Toolkit 3.0 FAQ*, (Last update: 19 November 2002). Retrieved: 24 March 2003, from http://www.globus.org/toolkit/gt3-faq.html.

Gong, L. (2001a), 'JXTA: A Network Programming Environment', *IEEE Internet Computing,* vol. 5, no. 3, pp. 88-95. Retrieved: 25 February 2003, from http://www.jxta.org/project/www/docs/mdejxta-paper.pdf.

Gong, L. (2001b), *Project JXTA: A Technology Overview*, 29 October 2009. Retrieved: 25 February 2003, from http://www.jxta.org/project/www/docs/jxtaview_01nov02.pdf.

GRAIL, *GRAIL Project page*. Retrieved: 3 October 2003, from http://grail.sdsc.edu/main_pages/projects.html.

Gupta, A. & Kumar, V. (1993), 'Performance Properties of Large Scale Parallel Systems', *Journal of Parallel and Distributed Computing,* vol. 19, no. 3, pp. 234-244. Retrieved: 5 April 2004, from http://www-users.cs.umn.edu/~kumar/papers/performance.ps.

Gustafson, J. L. (1988), 'Reevaluating Amdahl's Law', *Communications of the ACM,* vol. 31, no. 5, pp. 532 - 533. Retrieved: 2 December 2002, from http://www.scl.ameslab.gov/Publications/Gus/AmdahlsLaw/Amdahls.pdf.

Ibbett, R. N. & Topham, N. P. (1989), *Architecture of High Performance Computers*, Macmillan Education Ltd, London, Great Britain.

IBM, *IBM TSpaces Programmer's Guide*. Retrieved: 17 February 2003, from http://www.almaden.ibm.com/cs/TSpaces/html/ProgrGuide.html.

IBM, *IBM TSpaces User's Guide*. Retrieved: 17 February 2003, from http://www.almaden.ibm.com/cs/TSpaces/html/UserGuide.html.

IBM (2003), 'What is TSpaces?' in *TSpaces*. Retrieved: 10 February 2003, from http://www.almaden.ibm.com/cs/TSpaces/intro.html.

Innovations Softwaretechnologie GmbH (2004), *visual rules*. Retrieved: 19 April 2004, from http://www.visual-rules.de/en/pdf/en_visual_rules.pdf.

ISDA (2004), *FpML: The XML Standard for Swaps, Derivatives and Structured Products*. Retrieved: 20 April 2004, from http://www.fpml.org/.

ISO (2004), *ISO 15022 format homepage*, (Last update: 16 June 2004). Retrieved: 20 April 2004, from http://www.iso15022.org/.

iWay Software (2003), *iWay XML Transformation Engine (iXTE)*. Retrieved: 28 January 2003, from http://www.iwaysoftware.com/products/xmltransformationserver.html.

Jacob, B. (2003), *Grid computing: What are the key components?*, June 2003. Retrieved: 12 March 2003, from ftp://www6.software.ibm.com/software/developer/library/gr-overview.pdf.

JBoss, *JBoss :: Professional Open Source*. Retrieved: 22 April 2003, from http://www.jboss.org/.

Joergensen, M. (2001), *XSLTC Documentation*, 13 December 2001. Retrieved: 16 September 2004, from http://xml.apache.org/xalan-j/xsltc/.

jythonc, *Compiling Python Source to Real Java Classes*. Retrieved: 16 September 2004, from http://www.jython.org/docs/jythonc.html.

Kaiser, T., *An Overview of Parallel Computing Scalable Architectures and their Software.* Retrieved: 20 April 2004, from http://www.navo.hpc.mil/pet/Video/Courses/SDSC/PDF/sdsc_sw1.pdf.

Karora (2003), *XMLConnector Overview*. Retrieved: 29 January 2003, from http://www.karora.com/xmlconnector/xmlconnoview.htm.

Krauter, K., Buyya, R. & Maheswaran, M. (2002), 'A Taxonomy and Survey of Grid Resource Management Systems', *Software: Practice & Experience,* vol. 32, no. 2, pp. 135 - 164. Retrieved: 12 March 2003, from http://choices.cs.uiuc.edu/2k/papers/Internal/related/grid/grid-taxonomy.pdf.

Le, T. T. & Huu, T. C. (1997), *Advances in Parallel Computing For the Year 2000 and Beyond*. Retrieved: 20 April 2004, from http://www.vacets.org/vtic97/ttle.htm.

Legion (2001), *Legion: A Worldwide Virtual Computer*, (Last update: 20 June 2001). Retrieved: 19 March 2003, from http://legion.virginia.edu/.

Lehman, T. J. (2002), 'Current Status of TSpaces at IBM -- from the horse's mouth', TSpaces discussion list, 29 August 2002. Retrieved: 10 February 2003, from http://www.topica.com/lists/tspaces/read/message.html?sort=d&mid=905132325&start=112.

Lehman, T. J., Cozzi, A., Xiong, Y., Gottschalk, J., Vasudevan, V., Landis, S., Davis, P., Khavar, B. & Bowman, P. (2001), 'Hitting the distributed computing sweet spot with TSpaces', *Computer Networks,* vol. 35, no. 4, pp. 457 - 472. Retrieved: 10 February 2003, from http://www.almaden.ibm.com/cs/TSpaces/papers/ComputerNetworks.pdf.

Lehman, T. J., McLaughry, S. W. & Wyckoff, P. (1999), 'T Spaces: The Next Wave', in *Proceedings of the 32nd Hawaii International Conference on System Sciences*, Hawaii, USA. Retrieved: 10 February 2003, from http://www.almaden.ibm.com/cs/TSpaces/papers/Cluster.ps.Z.

Liu, H., Weng, S. & Sun, W. (2001), *Cache, Matrix Multiplication, and Vector*. Retrieved: 21 June 2004, from http://www.cs.umd.edu/class/fall2001/cmsc411/proj01/cache/matrix.html.

Livny, M. (2002), *Welcome and Condor Project Overview*. Retrieved: 25 March 2003, from http://www.cs.wisc.edu/condor/presentations/PC-2002/miron.ppt.

Natrajan, A., Humphrey, M. A. & Grimshaw, A. S. (2001), 'Grids: Harnessing Geographically-Separated Resources in a Multi-Organisational Context', in *Proceedings of the 15th Annual Symposium on High Performance Computing Systems and Applications*, Ontario, Canada. Retrieved: 20 March 2003, from http://legion.virginia.edu/papers/HPCS01.pdf.

OAG (2003), *Open Applications Group*. Retrieved: 5 February 2003, from http://www.openapplications.org.

OASIS (2003), *OASIS Universal Business Language TC*. Retrieved: 29 May 2003, from http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ubl.

OGSI-WG (2003), *Open Grid Services Infrastructure (OGSI) Version 1.0*, 13 March 2003. Retrieved: 21 March 2003, from http://www.gridforum.org/ogsi-wg/drafts/draft-ggf-ogsi-gridservice-26_2003-03-13.pdf.

Ourusoff, N. (2003), *Tutorial on JSP & JSD*. Retrieved: 12 December 2002, from http://cisx2.uma.maine.edu/NickTemp/JSP&JSDLec/jsd.html.

PerCurrence (2000), *Core PerXML Technology*. Retrieved: 28 January 2003, from http://www.percurrence.com/products/xslfaq.html.

Plachy, O. (1997), *Parallel architectures*, 4 April 1997. Retrieved: 20 April 2004, from http://zikova.cvut.cz/parallel/diplom/node4.html.iso-8859-1.

Platform Computing (2003a), *Platform Computing - Products*. Retrieved: 25 September 2003, from http://www.platform.com/products/.

Platform Computing (2003b), *Platform JobScheduler*. Retrieved: 25 September 2003, from http://www.platform.com/PDFs/datasheets/Plt_JobScheduler_DS.pdf.

Platform Computing (2003c), *Platform LSF*. Retrieved: 25 September 2003, from http://www.platform.com/PDFs/datasheets/Plt_LSF_DS.pdf.

Platform Computing (2003d), *Platform Software and Services*. Retrieved: 25 September 2003, from http://www.platform.com/pdfs/datasheets/Plt_Products_Overview.pdf.

Prabhu, G. M. (2003), 'Principle of Locality', in *Computer Architecture Tutorial*. Retrieved: 20 June 2004, from http://www.cs.iastate.edu/~prabhu/Tutorial/CACHE/pr_locality.html.

Propylon (2003), *PropelXbi Product Data Sheet*. Retrieved: 29 May 2003, from http://www.propylon.com/products/datasheets/propelxbi-datasheet.pdf.

Quin, L. (2004), 'Frequently Asked Questions', in *What is XSL?*, 8 July 2004. Retrieved: 13 September 2004, from http://www.w3.org/Style/XSL/WhatIsXSL.html.

Quovadx (2003), *Ruple*. Retrieved: 11 February 2003, from http://www.roguewave.com/developer/tac/ruple/.

Redmond, T. & McGrath, S. (2002), *XPipe*. Retrieved: 27 January 2003, from http://www.propylon.com/news/events/barcelona02.ppt.

RosettaNet (2004), *PIPs*. Retrieved: 20 April 2004, from http://www.rosettanet.org/RosettaNet/Rooms/DisplayPages/LayoutInitial?Container=com.webridge.entity.Entity%5BOID%5B279B86B8022CD411841F00C04F689339%5D%5D.

SAP, *SAP INFO IDOC*. Retrieved: 20 April 2004, from http://www.sap.info/public/en/glossary.php4/list/Word-38053d4a8777d94a6_glossary/I.

SAX (2003), *SAX*. Retrieved: 29 May 2003, from http://www.saxproject.org/.

SETI@Home, *SETI@home: Search for Extraterrestrial Intelligence at home*. Retrieved: 10 February 2003, from http://setiathome.ssl.berkeley.edu/.

Shalom, N. (2002a), *Are You Ready for GRIDS?* Retrieved: 10 February 2003, from http://www.j-spaces.com/download/IJUG_GRIDS.pdf.

Shalom, N. (2002b), *JavaSpaces*. Retrieved: 10 February 2003, from http://www.j-spaces.com/download/IJUG_JavaSpaces.pdf.

Strain, E. (2003), *esa://Tuple Spaces*, 11 July 2002. Retrieved: 10 February 2003, from http://earl.strain.at/space/Tuple+Spaces.

Sun Microsystems, *Java System Message Queue General FAQs*. Retrieved, from http://wwws.sun.com/software/products/message_queue/faqs_message_queue.html.

Sun Microsystems, *Project JXTA*. Retrieved: 25 February 2003, from http://wwws.sun.com/software/jxta/.

Sun Microsystems (2001a), *How Sun Grid Engine, Enterprise Edition 5.3 Works*, November 2001. Retrieved: 26 September 2003, from http://wwws.sun.com/software/gridware/sgeee53/wp-sgeee/wp-sgeee.pdf.

Sun Microsystems (2001b), *Maximizing Computing Resources: Sun Grid Engine, Enterprise Edition Software*, December 2001. Retrieved: 26 September 2003, from http://wwws.sun.com/software/gridware/sgeee53/sgeee.pdf.

Sun Microsystems (2001c), *Project JXTA: An Open, Innovative Collaboration*, 25 April 2001. Retrieved: 25 February 2003, from http://www.jxta.org/project/www/docs/OpenInnovative.pdf.

Sun Microsystems (2001d), *Project JXTA: Java Programmer's Guide*. Retrieved: 25 February 2003, from http://www.jxta.org/docs/jxtaprogguide_final.pdf.

Sun Microsystems (2002a), *Distributed Resource Management: Sun Grid Engine Software*, April 2002. Retrieved: 26 September 2003, from http://wwws.sun.com/software/gridware/ds-gridware/sge.pdf.

Sun Microsystems (2002b), *Endorsed Standards Override Mechanism*. Retrieved: 17 September 2004, from http://java.sun.com/j2se/1.4.2/docs/guide/standards/index.html.

Sun Microsystems (2002c), *Grid Computing - optimizing the performances and resources of your team.*, June 2002. Retrieved: 25 September 2003, from http://wwws.sun.com/software/grid/Grid-brochure.pdf.

Sun Microsystems (2002d), *JavaSpaces Service Specification*, April 2002. Retrieved: 10 February 2003, from http://wwws.sun.com/software/jini/specs/js1_2_1.pdf.

Sun Microsystems (2002e), *Project JXTA Technology*, September 2002. Retrieved: 25 February 2003, from http://wwws.sun.com/software/jxta/JXTA5.pdf.

Sun Microsystems (2002f), *Sun Cluster Grid Architecture*, May 2002. Retrieved: 25 September 2003, from http://wwws.sun.com/software/grid/SunClusterGridArchitecture.pdf.

Sun Microsystems (2003a), *JavaSpaces Technology*. Retrieved: 10 February 2003, from http://java.sun.com/products/javaspaces/.

Sun Microsystems (2003b), *JXTA v2.0 Protocols Specification*. Retrieved: 25 March 2003, from http://spec.jxta.org/v1.0/docbook/JXTAProtocols.pdf.

Sun Microsystems (2004), *J2EE FAQ*, (Last update: 5 June 2003). Retrieved: 25 June 2004, from http://java.sun.com/j2ee/faq.html.

Sutcliffe, A. (1988), *Jackson System Development*, Prentice Hall International (UK) Ltd.

Traversat, B., Abdelaziz, M., Duigou, M., Hughly, J.-C., Pouyoul, E. & Yeager, B. (2002), *Project JXTA Virtual Network*, 5 February 2002. Retrieved: 25 February 2003, from http://www.jxta.org/project/www/docs/JXTAprotocols_01nov02.pdf.

Tvrdík, P. (2002), 'Analyza slozitosti paralelnich algoritmu', in *Paralelní systémy a algoritmy*, vol. 2, Vydavatelství ČVUT, Prague, Czech Republic, pp. 6-15.

Unicorn (2003), *Semantic Information Management*. Retrieved: 20 April 2004, from http://www.unicorn.com/semantics/sim.htm.

Verbeke, J., Nadgir, N., Ruetsch, G. & Sharpov, I., *Framework for Peer-to-Peer Distributed Computing in a Heterogeneous, Decentralized Environment*. Retrieved: 25 February 2003, from http://www.jxta.org/project/www/docs/mdejxta-paper.pdf.

Voicu, R. (2004), *Introduction, Overview, Parallel Architectures*, 7 January 2004. Retrieved: 20 April 2004, from http://www.comp.nus.edu.sg/~cs4231/cs4231-lec01.pdf.

W3C (1999), *XSL Transformations (XSLT) Version 1.0*, 19 November 1999. Retrieved: 29 May 2003, from http://www.w3.org/TR/xslt.

W3C (2000), 'CDATA Sections', in *Extensible Markup Language (XML) 1.0 (Second Edition)*, 6 October 2000. Retrieved: 16 September 2004, from http://www.w3org/TR/2000/REC-xml-20001006.

W3C (2002), *Document Object Model (DOM) homepage*. Retrieved: 28 January 2003, from http://www.w3.org/DOM/.

Weissman, J. B. (2002), *Research Summary - November 2002*, November 2002. Retrieved: 20 April 2004, from http://www-users.cs.umn.edu/~jon/res_stmt.pdf.

Wyckoff, P., McLaughry, S. W., Lehman, T. J. & Ford, D. A. (1998), 'T Spaces', *IBM Systems Journal*, vol. 37, no. 3, pp. 454-474. Retrieved: 10 February 2003, from http://www.research.ibm.com/journal/sj/373/wyckoff.html.

Xbeans (2003), *Xbeans*. Retrieved: 28 January 2003, from http://www.xbeans.org.

Zhao, B. Y. (1998), *TupleSpaces Revisited: Linda to TSpaces*, 13 July 1998. Retrieved: 10 February 2003, from http://www.cs.berkeley.edu/~ravenben/research/tuplespace/tuplespace.PPT.

APPENDICES

APPENDIX A

EFFICACY OF SDMP-SCATTER/GATHER APPROACH

Appendix A. Efficacy of SDMP-Scatter/Gather approach

In this appendix, we study the execution time of transformations when using the Scatter / Gather approach and point out the implications about the efficacy of its use. Furthermore we introduce formulas for respective gain and maximal number of beneficial processors.

To express the time of completion of the transformation in SDMP-Scatter/Gather system we developed following formula:

$$t_{SDMP-s/g}(n) = \max\left(t(ns), t\left(\frac{np}{k}\right)\right) + t_m(n,k)$$

Equ. 1 Time of completion of SDMP – scatter/gather scenario

$t(n)$	execution time depending on size of input
$t_m(n,k)$	time spent by doing maintenance operations (pre- and post-processing stage). In this case, it's scatter and gather stage.
n	size of input
p	parallel (scatterable) part of input (%)
s	$(1-p)$ serial (in-scatterable) part of input (%)
k	number of scattered segments

Note: $k \le P_{\max}$ where $P_{\max}$ is number of processors available to be added

The time taken for the transformation of any input of size n is determined as the sum of the longer of the times for processing the serial and the parallel parts and time spent on the maintenance operations which is the same regardless of ratio between parallel and serial parts. The completion time for processing the serial part depends on the portion of input, which needs to be processed serially and the size of the input. The timespan for the processing of the parallel part again depends on the portion of input to process, but as this part of the document is scattered into k parts, the input that is actually processed is k times smaller. We assume, that the maintenance time t_m is dependent on the size of input and the number of segments into which it divides the scatterable part of the document.

We presume, that in most cases, we can neglect t_m as it doesn't considerably contribute to the total time of processing. The equation then simplifies to

$$t_{SDMP-s/g}(n) = \max\left(t(ns), t\left(\frac{np}{k}\right)\right)$$

To give an example of the use of this formula, let's say that we have 100KB document: 60% of which is parallelizable; normal processing of a 100KB document takes 10 seconds; and the time function is linear. The time function then is $t(n) = \frac{10}{100KB}n$, and if we use two additional processors the total execution time equals

$$\begin{aligned} t(100KB) &= \max\left(t(100KB\cdot 0.4), t\left(\frac{100KB\cdot 0.6}{2}\right)\right) = \\ &= \max\left(\frac{10}{100KB}100KB\cdot 0.4, \frac{10}{100KB}\frac{100KB\cdot 0.6}{2}\right) = \\ &= \max(10\cdot 0.4, 10\cdot 0.3) = 4 \end{aligned}$$

The execution time decreases to 4 seconds, but as can be seen from equation above it can't ever decrease more, as now the processing time for the serial part determines the result.

To see how much we gained by using the SDMP approach we define gain as a new variable $G_{A/B}$ stating how much less time the transformation takes in system A compared to B.

$$G_{A/B} = 1 - \frac{t_A}{t_B} = \frac{t_B - t_A}{t_B} \quad (A.1)$$

For example, when the transformation takes three times less in system A ($t_A = 100$) compared to B ($t_B = 300$) the gain is $G_{A/B} = 1 - 1/3 = 66\%$.

In the literature about parallel processing, there is another variable used. This variable S is the speed-up which states how much faster the new approach is (Gustafson 1988; Tvrdík 2002). It is defined by $S_{A/B} = \frac{t_B}{t_A}$, what equals $S_{A/B} = \frac{1}{1-G_{A/B}}$. In our example $S_{A/B} = \frac{1}{\frac{1}{3}} = 3$ i.e. the transformation in system A is three times faster.

In the following text, indices of G will be omitted when the meaning of the symbol is apparent from the context.

The time of completion can be then expressed using the variables defined above as follows:

$$t = (1-G)\,t_{SDSP} = \frac{t_{SDSP}}{S}$$

To express gain earned by employing the SDMP-s/g approach compared with the SDSP approach we use previously defined Equ 1 and derive following formula:

$$G_{SDMP/SDSP} = 1 - \frac{\max\left(t(ns) + t\left(n\frac{p}{k}\right)\right) + t_m(n,k)}{t(n)}$$

Equ. 2 General SDMP-s/g / SDSP gain definition

When we omit the maintenance cost (which can be done when the parallelizable part of the document is reasonably large) we get:

$$G_{SDMP/SDSP} = 1 - \frac{\max\left(t(ns) + t\left(n\frac{p}{k}\right)\right)}{t(n)}$$

Applied to the example of 100KB document we used earlier in the text, the gain when using two additional processors is

$$G_{SDMP/SDSP} = 1 - \frac{\max\left(t(n \cdot 0.4) + t\left(n \cdot 0.6/2\right)\right)}{t(n)} = 1 - \frac{4}{10} = 0.6 = 60\%.$$

In many cases, the time function can be expressed as $t(n) = cn^m$. It means that the completion time is given by the power m of the input size multiplied by an arbitrary constant c. The actual time function may be more complicated, but we believe that for many cases this approximation is sufficient.

For $t(n) = cn^m$ we can express the gain as:

$$G = 1 - \frac{\max\left(c(ns)^m, c\left(n\frac{p}{k}\right)^m\right)}{cn^m} = 1 - \frac{cn^m \max\left(s^m, \frac{p^m}{k}\right)}{cn^m} =$$
$$= 1 - \max\left(s^m, \frac{p^m}{k^m}\right)$$

It says, in short, that the gain increases until the processing time of one parallel portion of the data is less than or equal to the processing time of the serial part. After this point of saturation, the gain stays constant as the time of completion is limited by the time of processing the serial part.

The maximal gain is upper-bounded by the size of the serial part and its value can be obtained from examining the limit case when the number of documents grows to infinity. As gain is $G = 1 - \max\left(s^m, \frac{p^m}{k^m}\right)$ its limit case is $G_{\max} = \lim_{k \to \infty} G = 1 - s^m$. This corresponds to the situation when the processing time of the parallel parts diminishes to zero and the only data to process is the serial part. The maximal gain in our example is $G_{\max} = 1 - 0.4 = 0.6 = 60\%$, which again shows that in the example that we used earlier, the obtained gain is the maximal possible and the higher gain can't be achieved by adding more processors.

When the execution time of the parallel parts reaches the execution time of the serial part we get to saturation point, which indicates the maximal number of beneficial processors $k_{\max}$ whose addition brings any gain. Every added processor after this number doesn't bring any gain and only increases maintenance time.

As saturation occurs when the time of processing of the parallel part catches up with the time of processing of the serial part, we infer that $k_{\max}$ can be obtained from the equality of these two times:

$$s^m = \frac{p^m}{k_{\max}^m} \Leftrightarrow k_{\max}^m = \frac{p^m}{s^m} \Rightarrow k_{\max} = \sqrt[m]{\frac{p^m}{s^m}} \Rightarrow$$
$$\Rightarrow k_{\max} = \left\lceil \frac{p}{s} \right\rceil$$

In the last step of our derivation, we applied ceiling function to the right hand side of the equation as value of $k_{\max}$ must be an integer.

In our previous example, p is 0.6 and s is 0.4. This tells us straight away that the maximal number of beneficial processors is $k_{\max} = \lceil 0.6/0.4 \rceil = \lceil 1.5 \rceil = 2$.

The interesting point is that $k_{\max}$ is independent of the order of the function (m) and its multiplicative constant (c). Therefore, the maximal number of beneficial processors stays the same when the time function (processing algorithm) changes.

There can be a case when processing of the serial part has a different speed from the processing of the parallel parts, e.g. the serial part can be just copied to the output without any further processing. In this case, we introduce two different time functions $t_s(n) = c_s n^m$ (for the serial part) and $t_p(n) = c_p n^m$ (for the parallel parts). We furthermore set $c = c_p$ as the processing speed of the whole document and the processing speed of its parallel parts is usually the same (the algorithm usually doesn't change). With those additions, we derive expression for gain as following:

$$G = 1 - \frac{\max\left(c_s (ns)^m, c_p \left(n\frac{p}{k}\right)^m\right)}{c_p n^m} = 1 - \frac{\max\left(c_p \frac{c_s}{c_p}(ns)^m, c_p\left(n\frac{p}{k}\right)^m\right)}{c_p n^m} =$$

$$= 1 - \frac{c_p n^m \max\left(\frac{c_s}{c_p}s^m, \frac{p^m}{k}\right)}{c_p n^m} =$$

$$= 1 - \max\left(\frac{c_s}{c_p}s^m, \frac{p^m}{k^m}\right)$$

This shows that, as the time of processing the serial part decreases (less time is spent on serial part), the gain increases and the point of saturation shifts to larger numbers.

The maximal number of beneficial processors is then changed to

$$k_{\max}^m = \frac{c_p}{c_s}\frac{p^m}{s^m} \Leftrightarrow k_{\max} = \sqrt[m]{\frac{c_p}{c_s}\frac{p^m}{s^m}} \Rightarrow$$

$$\Rightarrow k_{\max} = \left\lceil \sqrt[m]{\frac{c_p}{c_s}}\frac{p}{s} \right\rceil$$

The maximal gain changes, accordingly, to

$$G_{\max} = \lim_{k\to\infty} G = \lim_{k\to\infty}\left(1 - \max\left(\frac{c_s}{c_p}s^m, \frac{p^m}{k^m}\right)\right) = 1 - \frac{c_s}{c_p}s^m$$

To illustrate this situation, when the speed of processing of the serial part differs from the execution time of the processing of the parallel part, we use the earlier example of 100KB document with 60% of parallelizable content. Let's say that the serial part of the document is just copied and thus its processing time is ten times shorter (processing of 100KB would take 1 second). The constant of the processing of the serial part is $c_s = 1/100KB$ and for the parallel processing the constant $c_p = 10/100KB$. The gain is then

$$G = 1 - \frac{\max\left(1/100KB \cdot 100KB \cdot 0.4, 10/100KB \cdot 100KB \cdot \frac{0.6}{2}\right)}{10/100KB \cdot 100KB} =$$

$$= 1 - \frac{\max(1 \cdot 0.4, 10 \cdot 0.3)}{10} =$$

$$= 1 - \frac{10 \cdot 0.3}{10} = 0.7 = 70\%$$

As processing of the serial part takes a shorter time now, it is the parallel processing time which prevails and the gain is raised to seventy percent. The maximal gain also rises to

$$G_{\max} = 1 - \frac{c_s}{c_p} s = 1 - \frac{1/100KB}{10/100KB} 0.4 = 0.96 = 96\%$$

and the maximal number of beneficial processors rises to

$$k_{\max} = \left\lceil \sqrt{\frac{10/100KB}{1/100KB}} \cdot \frac{0.6}{0.4} \right\rceil = \lceil 10 \cdot 1.5 \rceil = 15$$

This demonstrates the earlier observation that with decreasing time spent on the serial part, $k_{\max}$ becomes greater and, in the limit case where the serial part is omitted completely ($c_s = 0$), $k_{\max}$ grows to infinity.

The equations we introduced in this chapter give very important information about the gain that can be obtained and the maximal number of beneficial processors. It leads to the following three conclusions.

1) **The higher the order of the time function (m), the greater the savings that are achieved by using the scatter/gather approach.**

This follows from the gain definition. We expressed gain as $1-\max\left(s^m, \frac{p^m}{k^m}\right)$. When the number of processors is less than $k_{\max}$, the parallel portion dominates and the expression is $1-\frac{p^m}{k^m}$. The fraction $\frac{p}{k}$ is less than one, hence with a higher order of the time function it is powered by a higher exponent, thus it results in a smaller number by which the value one is reduced. This fact is demonstrated by the following graphs, which show the gain and the speed-up for documents with different amounts of parallelizable portions and for the time functions $t(n)=cn$, $t(n)=cn^2$ and $t(n)=cn^3$.

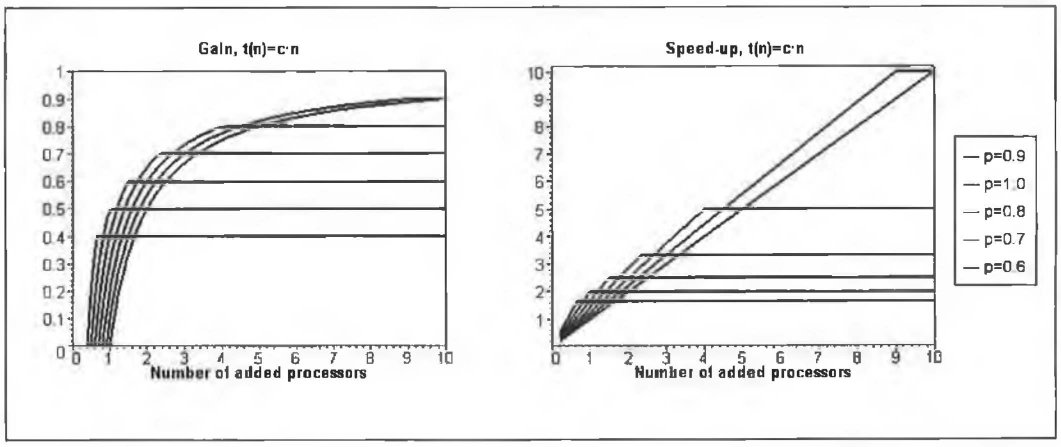


Fig. A.1 Gain and Speed-up for t(n)=cn

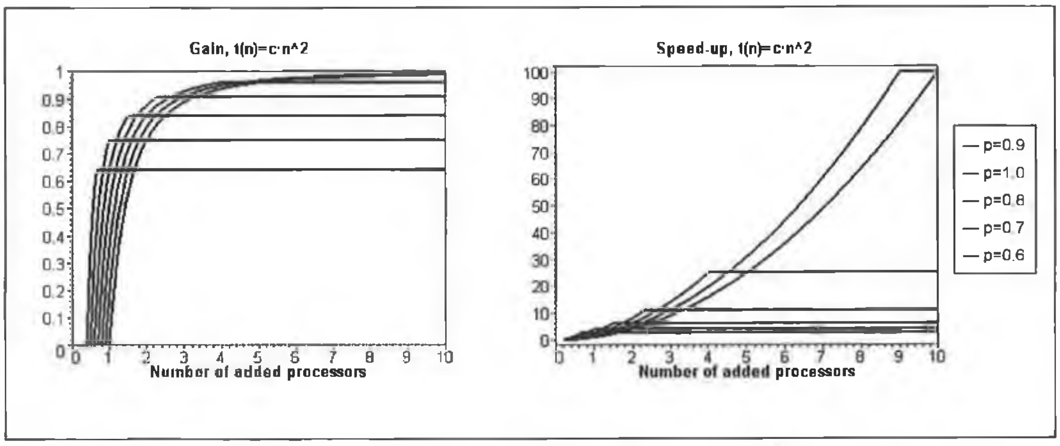


Fig. A.2 Gain and Speed-up for t(n)=cn^2

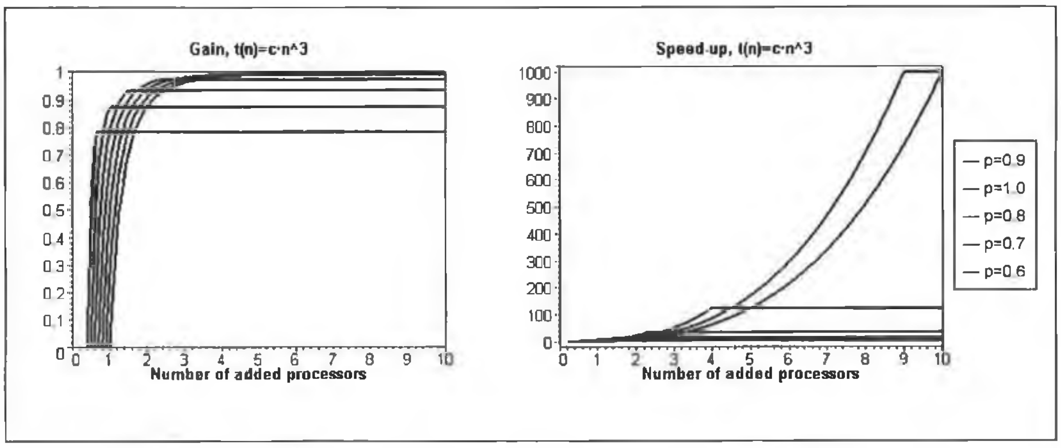


Fig. A.3 Gain and Speed-up for t(n)=cn^3

The difference in gain increase for the different orders of the time function can be clearly seen in the graphs, above. In practice, the first two cases, i.e. with linear and quadratic time functions, are the most common.

2) Number of processors bringing gain is limited and independent of order of time function

As can also be seen from the above graphs, there is a point of saturation after which adding processors does not bring any gain because the processing time of the whole document is set by the processing time of the serial part (which stays constant for any number of processors).

For the time function expressed as $t(n) = cn^m$, this number is

$$k_{\max} = \left\lceil \frac{p}{s} \right\rceil$$

When the speed of the serial and parallel parts varies, and is set by the coefficients c_s and c_p, then $k_{\max}$ is given by:

$$k_{\max} = \left\lceil \sqrt[m]{\frac{c_p}{c_s}\frac{p}{s}} \right\rceil$$

3) Gain from parallelization can not ever be grater than $1-\frac{c_s}{c_p}s^m$

As we defined gain as $G=1-\max\left(\frac{c_s}{c_p}s^m,\frac{p^m}{k^m}\right)$, its limit case is $\lim_{k\to\infty}G=1-\frac{c_s}{c_p}s^m$.

This corresponds to the situation where the processing time of the parallel parts diminishes to zero and the only data to process is the serial part. It is theoretically the greatest achievable gain. This gain can't be ever reached: at first, because we never have an infinite number of processors; and, secondly, because the maintenance overhead would outweigh the obtained gain.

Another way to look at this conclusion is to state that the speed-up cannot ever be greater than $\frac{1}{\frac{c_s}{c_p}s^m}=\frac{c_p}{c_s s^m}$. A variation of this statement is known as Amdahl's law in area of parallel processing (Gustafson 1988).

□

All previous calculations were performed for time functions in the form $t(n)=cn^m$. If, for any reason, a more precise expression of the time function is needed, then we lose the advantage of having the gain independent of the document size. This does not limit the possibility of evaluating the convenience of the scatter/gather approach, though.

In this case, it is necessary to use the general gain definition, Equ. 2

$$G=1-\frac{\max\left(t(ns),t\left(n\frac{p}{k}\right)\right)+t_m(n,k)}{t(n)}$$

and we need to display the gain as dependent on the size of the document for different numbers of processors. Example graphs for the time functions $t(n)=\log(n)$ and $t(n)=n\log(n)$ follow:

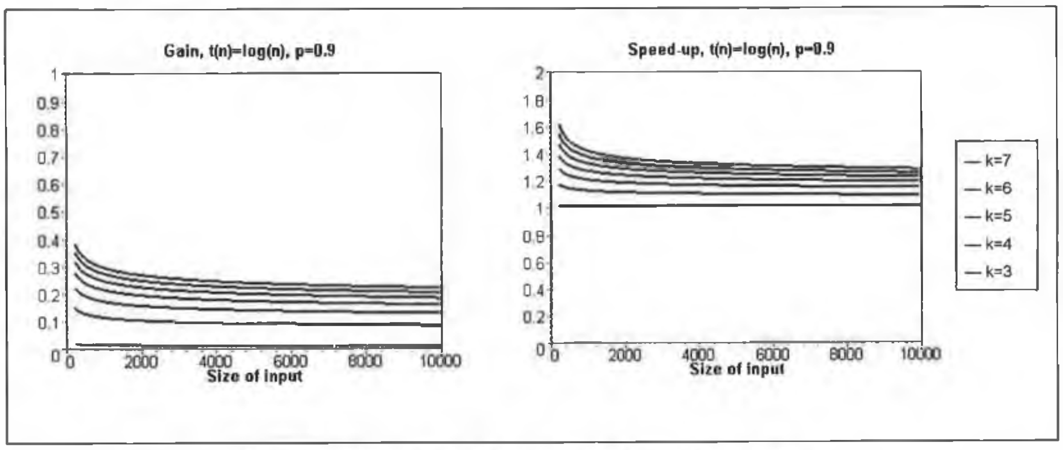


Fig. A.4 Gain and Speed-up for t(n)=log(n)

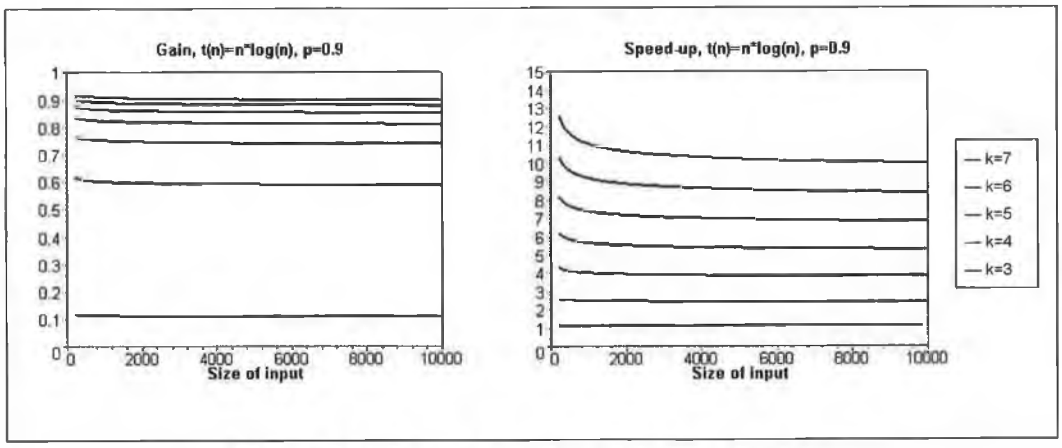


Fig. A.5 Gain and Speed-up for t(n)=nlog(n)

These two graphs demonstrate that scatter/gather delivers gain for both of these time functions, but with decreasing efficiency with growing size of input.

In cases where the parallelizable part of a document is small or adding another processor earns just a small or no gain, the amount of time spent on maintenance should be taken into consideration. The time spent on communication and processing overhead might outweigh the gain earned by employing more processors.

We can conclude our examination into following five points:

1) The higher the order of the time function, the bigger savings achieved by using the scatter/gather approach

2) The gain of scatter/gather approach is $G = 1 - \max\left(s^m, \frac{p^m}{k^m}\right)$ for $t(n) = c \cdot n^m$

   and the time of completion is $t = \max\left(s^m, \frac{p^m}{k^m}\right) \cdot t_{SDSP}$

3) The maximum number of beneficial processors is expressed by $k_{\max} = \left\lceil \frac{p}{s} \right\rceil$.

   When time of serial part processing is $t_s(n) = c_s \cdot n^m$ and time of parallel part $t_p(n) = c_p \cdot n^m$ then $k_{\max} = \left\lceil \sqrt[m]{\frac{c_p}{c_s}} \cdot \frac{p}{s} \right\rceil$.

   When $c_s = c_p = c$, $k_{\max}$ is independent of the order of the time function and its multiplicative constant.

4) The gain cannot ever be greater than $1 - \frac{c_s}{c_p} s^m$. The speed-up can't ever be greater than $\frac{c_p}{c_s \cdot s^m}$ for $t(n) = c \cdot n^m$

5) The convenience of the scatter/gather approach for transformations when $t(n) \neq c \cdot n^m$ can be easily seen from the graph of the gain with respect to the size of the input

Appendix B

Efficacy of MDMP-bulk aware processing

Appendix B. Efficacy of MDMP-bulk aware processing

In this part, we study the execution time of the transformations of MDMP-bulk aware processing and introduce formulae expressing gain obtained by using this approach.

We express the time of completion of the transformation in MDSP-bulk aware approach as a sum of the timespans of the pre-processing, core-processing and post-processing stages:

$$t(x) = t_{pre}(x) + t_{core}(x) + t_{post}(x)$$

Where x is the number of documents in a batch and $t_{pre}, t_{core}, t_{post}$ are the times of completion of the pre-, core- and post-processing stages. Again, we consider the processed documents to be a group of 'average documents' as defined in section 2.5.

As mentioned in chapter 2, the time savings in the MDSP-batch aware approach are achieved by minimising the pre- and post-processing stage. This is reflected in how $t_{pre}(x)$ and $t_{post}(x)$ are defined.

When the pre- and post-processing stages can be executed just once for the whole batch (e.g. opening and closing internet connection), then we define $t_{pre}(x)$ and $t_{post}(x)$ as constants:

$$t_{pre}(x) = c_{pre}$$

$$t_{post}(x) = c_{post}$$

In other cases, there is an additional small amount of work to be done for every document that is part of the pre- and post-processing stages (e.g. loading and saving a file to disk). We then define the time functions as follows:

$$t_{pre}(x) = c_{pre} + c_{pre2}x$$
$$t_{post}(x) = c_{post} + c_{post2}x$$

This states that work spent on pre- and post-processing is linearly dependent on the number of documents being processed. The constants c_{pre2} and c_{post2} are usually very small. The core-processing stage does not change when processing multiple documents.

Using previous definitions, we define gain of the MDSP-bulk aware approach as:

$$G_{\frac{MDSP-bulk\ aware}{MDSP-bulk\ unaware}} = 1 - \frac{t_{MDSP-ba}(x)}{t_{MDSP-bua}(x)} = 1 - \frac{t_{MDSP-ba}(x)}{xt_{MDSP-bua}(1)} =$$
$$= 1 - \frac{t_{pre}(x) + xt_{core}(1) + t_{post}(x)}{x(t_{pre}(1) + t_{core}(1) + t_{post}(1))}$$

The definition of gain G is the same as in (A.1). It states how much less time the transformation takes in new system compared to the old one.

When we combine t_{pre} and t_{post} to t_m, the expression simplifies to

$$G = 1 - \frac{t_m(x) + xt_{core}(1)}{x(t_m(1) + t_{core}(1))} = \frac{xt_m(1) - t_m(x)}{x(t_m(1) + t_{core}(1))} =$$
$$= \frac{xt_m(1) - t_m(x)}{xt(1)}$$

where $t_m(x) = t_{pre}(x) + t_{post}(x)$ and $t_{core}(x)$ are times of completion of the processing stages when processing x average documents of size n. The variable $t(x)$ is the time of completion of the whole transformation.

This equation shows apparent fact that the amount of gain we get is determined by the difference of times of the completion of x pre- and post-processing stages of the SDSP system and the pre- and post-processing stages of the MDSP system when processing x documents. In other words, as we reduce the overhead of the maintenance stages the gain is increased.

It also shows that if the time of completion of the core-processing stage is significantly greater than the maintenance time, then the whole gain approaches zero and is negligible.

To illustrate the given formula for gain, let's consider the following example. Suppose we process x documents of average size 100KB. Their processing consists of loading them from the disk, downloading the appropriate DTD's to validate them, transforming them and saving them back to disk again. The pre-processing stage consists of loading the files from disk, which takes an average 0.2 seconds per file and downloading the DTD which takes 4 seconds.

$$t_{pre}(x) = 0.2 \cdot x + 4\,.$$

The core-processing is the actual transformation which takes 10 seconds on average and so core-processing time function is $t_{core}(x) = 10 \cdot x$. Finally, the post-processing stage is saving the documents to disk, what again takes 0.2 seconds per file. $t_{post}(x) = 0.2 \cdot x$.

The maintenance time, which comprises the pre- and post-processing stages, can then be expressed as $t_m(x) = t_{pre}(x) + t_{post}(x) = 4 + 0.4 \cdot x$ and the total execution time as $t(x) = t_{core}(x) + t_m(x) = 10 + 4 + 0.4x = 14 + 0.4x$

The gain of the batch aware processing can be then expressed as:

$$G = \frac{x \cdot t_m(1) - t_m(x)}{x \cdot t(1)} = \frac{x \cdot (4+0.4) - (4 + 0.4 \cdot x)}{x \cdot (14 + 0.4)} = \frac{4x - 4}{14.4x} = \frac{4}{14.4}\left(1 - \frac{1}{x}\right)$$

For 10 documents, the gain evaluates to 25%, for 50 documents to 27% and for 100 documents to 27.5%.

Finally, the MDSP – batch aware gain formula shows the maximum gain that can be obtained. It can't be calculated in general for $t_m(x)$, because dependence of t_m on x isn't known, but we presume that in most cases $t_m(x)$ can be expressed, or sufficiently approximated as:

$$t_m(x) = c_m + c_{m2}x$$

The gain then simplifies to:

$$G = \frac{xt_m(1) - t_m(x)}{xt(1)} = \frac{x(c_m + c_{m2}) - (c_m + c_{m2}x)}{xt(1)} = \frac{(x-1)c_m}{xt(1)} = \\ = \frac{c_m}{t(1)}\frac{x-1}{x}$$

The maximum gain then is:

$$G_{\max} = \lim_{x\to\infty} G = \lim_{x\to\infty} \frac{c_m}{t(1)}\frac{x-1}{x} = \lim_{x\to\infty} \frac{c_m}{t(1)}\left(1 - \frac{1}{x}\right) = \frac{c_m}{t(1)}.$$

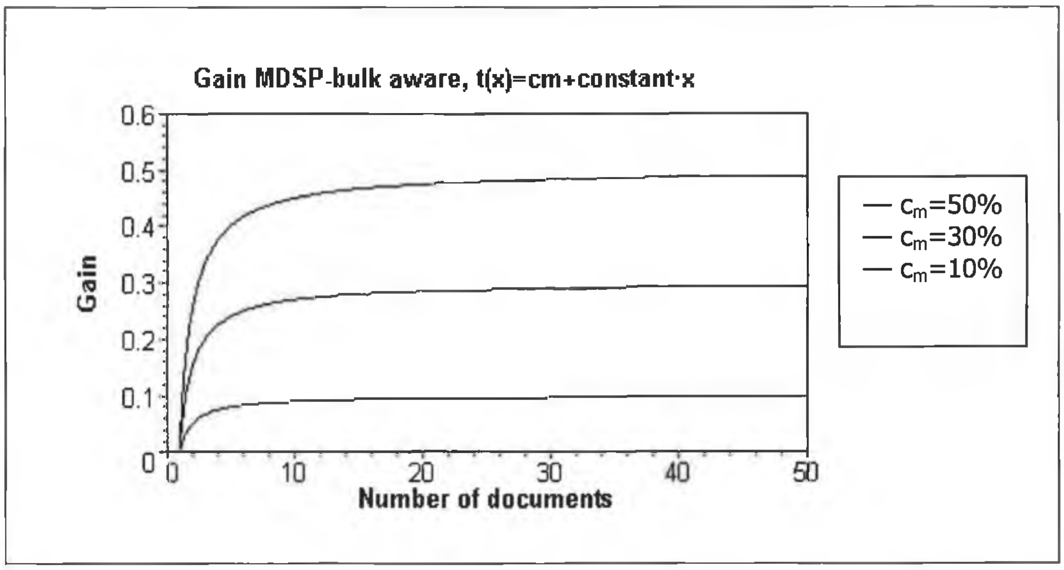
where c_m is the length of the join (shared) section of the transformation and $t(1)$ is the length of the whole transformation of one average document (which contains c_m as part of it).

This fraction in fact expresses the percentage amount of the joint section of the transformation in comparison to the length of the whole transformation. This joint section would theoretically diminish if the number of documents were infinite.

Similar to the SDMP-Scatter/Gather approach, this maximal gain cannot ever be reached, because there is never an infinite number of documents to process.

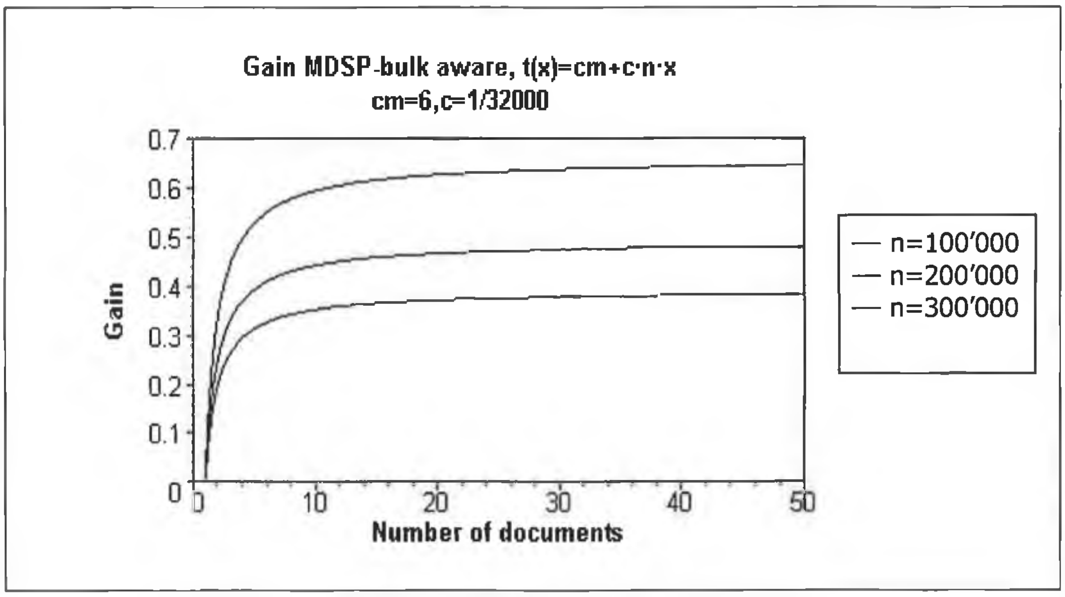
In the example we used earlier, t_m was expressed in the form $t_m(x) = 4 + 0.4 \cdot x$. The constant c_m is 4 and the time of processing one document $t(1) = 14.4$. The maximum gain then evaluates to $G_{\max} = \frac{c_m}{t(1)} = \frac{4}{14.4} = 0.278 = 27.8\%$. It's apparent that the gain obtained for 100 documents (27.5%) could not be raised much higher by increasing the number of processed documents.

The dependence of the amount of the gain on the joint segment size is demonstrated by the following graph. It shows that the larger the joint section is, compared to the whole transformation, the bigger the time savings are.



The time function $t(x)$ is considered constant in this graph. This shows the gain processing of an average document with the linear time function.

Alternatively, the two following graphs demonstrate the dependence of the gain achieved for individual input documents on the properties of the documents, namely, the order of the time function and the size of the document.



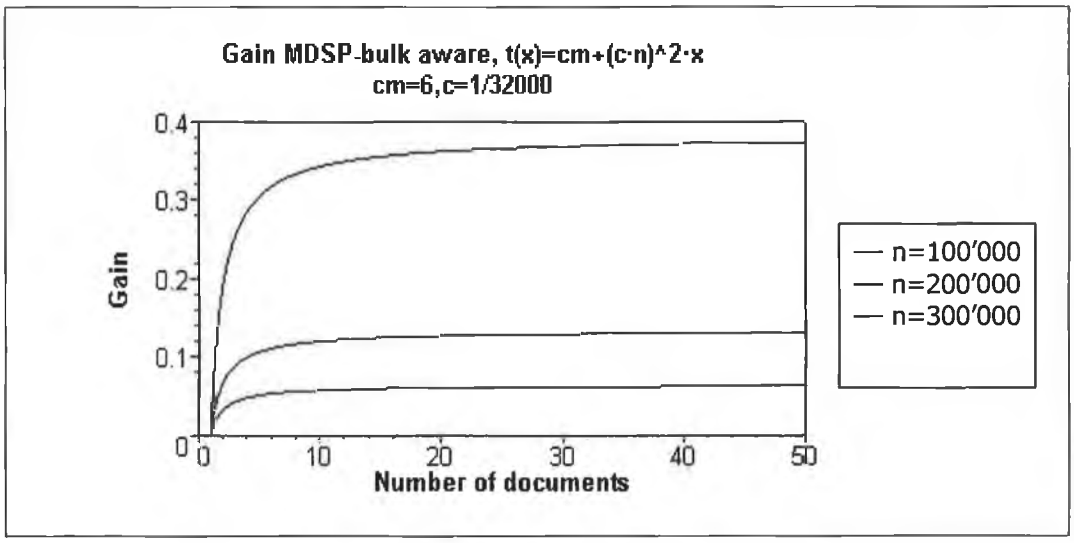


Fig. B.1 Gain in MDSP-bulk aware scenario

These two graphs demonstrate two other facts about the MDSP-bulk aware approach.

Firstly, when the size of the document increases, the obtained gain lessens as the ration of the joined section and the whole transformation reduces towards zero. Secondly, the higher the order of the time function is, the smaller the gain obtained. This is implied by the increased time of completion with the higher order of time function and consequential diminution of the ratio of the joined section and the whole transformation.

Our findings can be summarised in the following seven points:

1) The greater the joined (shared) section is, the greater the time savings are
2) The greater the size of document is, the lesser the time savings are
3) The higher order of the time function is, the lesser the time savings are

When the time function of the pre- and post-processing stages can be expressed as $t_{pre}(x)+t_{post}(x)=t_m(x)=c_m+c_{m2}x$ the following 3 points (4,5,6) are valid:

4) The gain is:

$$G=\frac{c_m}{t(1)}\frac{x-1}{x}$$

or

$$G=\frac{c_m}{t_m(1)+t_{core}(1)}\frac{x-1}{x}$$

where $t(1)$ is the time of processing of one average document and c_m is the time of completion of it's joined (shared) section.

5) The maximum gain is $G_{\max}=\frac{c_m}{t(1)}$

6) Because the gain function is always in the form $c\frac{x-1}{x}$, the shape of the function is always the same, only the upper limit of reachable gain changes for different time functions.

Therefore, the percentage of the gain reached by processing a certain number of documents is always the same irrespective of the time function.
The following table shows how much of the possible gain is obtained when processing at least x documents:

No of documents ≥	% of possible gain reached
3	66 %
4	75 %
10	90 %
20	95 %

7) The general gain expression for $t_m(x)$ is

$$G = \frac{xt_m(1) - t_m(x)}{xt(1)}$$

Appendix C

Selected Parallel Processing Graphs

Appendix C. Selected Parallel Processing Graphs

This appendix contains large versions of selected figures that are referred to in the text.

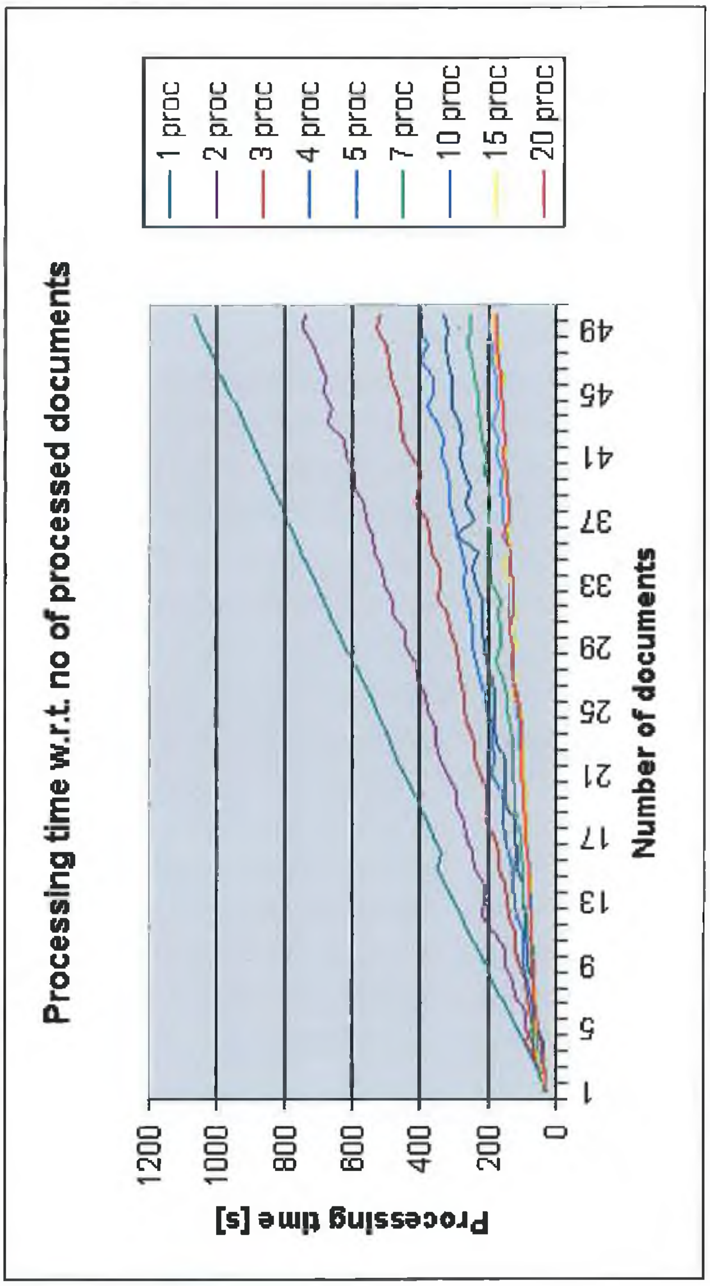


Fig. C.1 The measured parallel processing times with respect to the number of documents (Fig. 9.21)

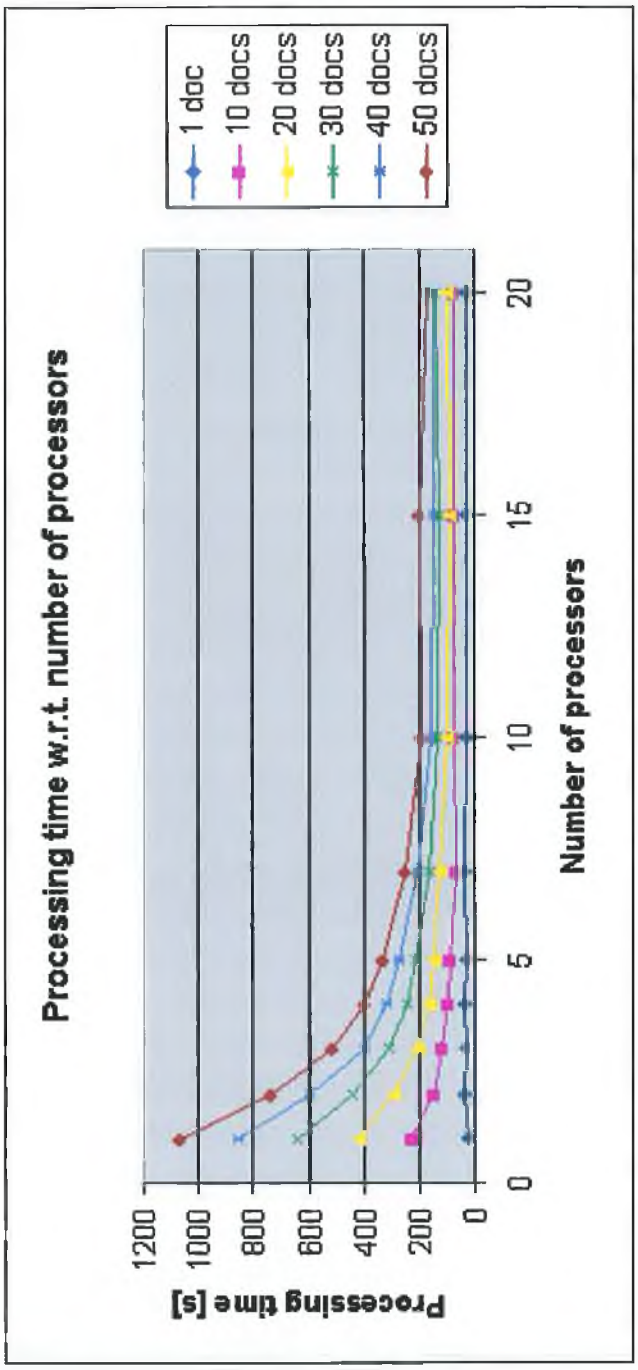


Fig. C.2 Measured parallel processing time with respect to number of processors (Fig. 9.22)

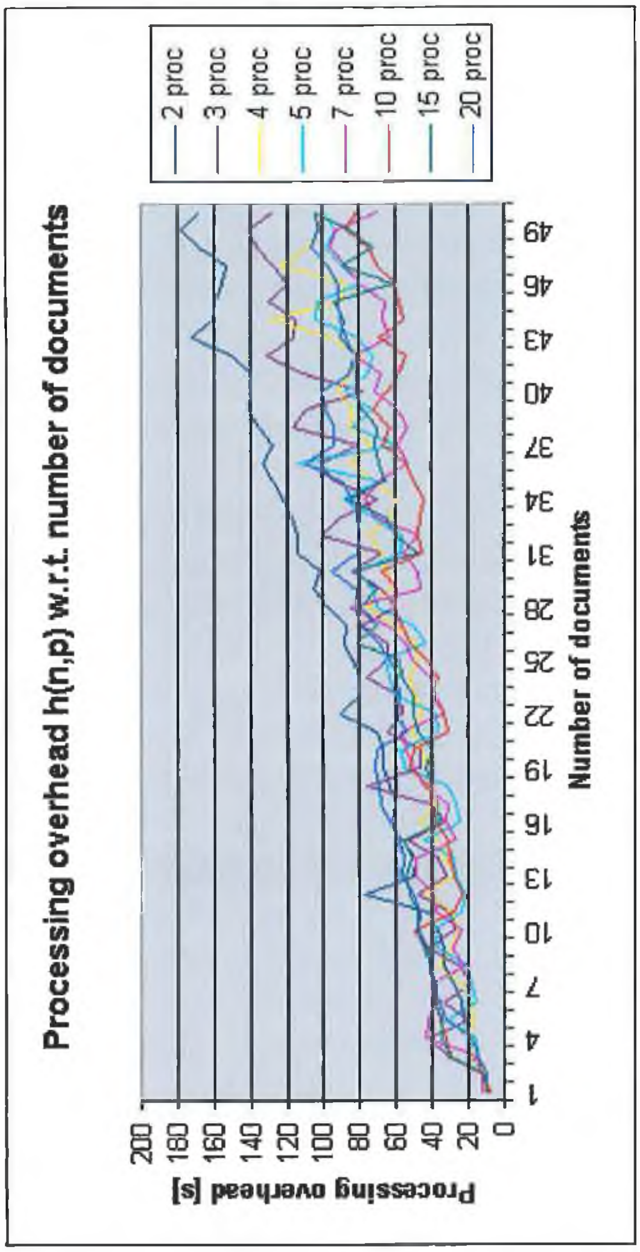


Fig. C.3 Processing overhead with respect to number of documents (Fig. 9.23)

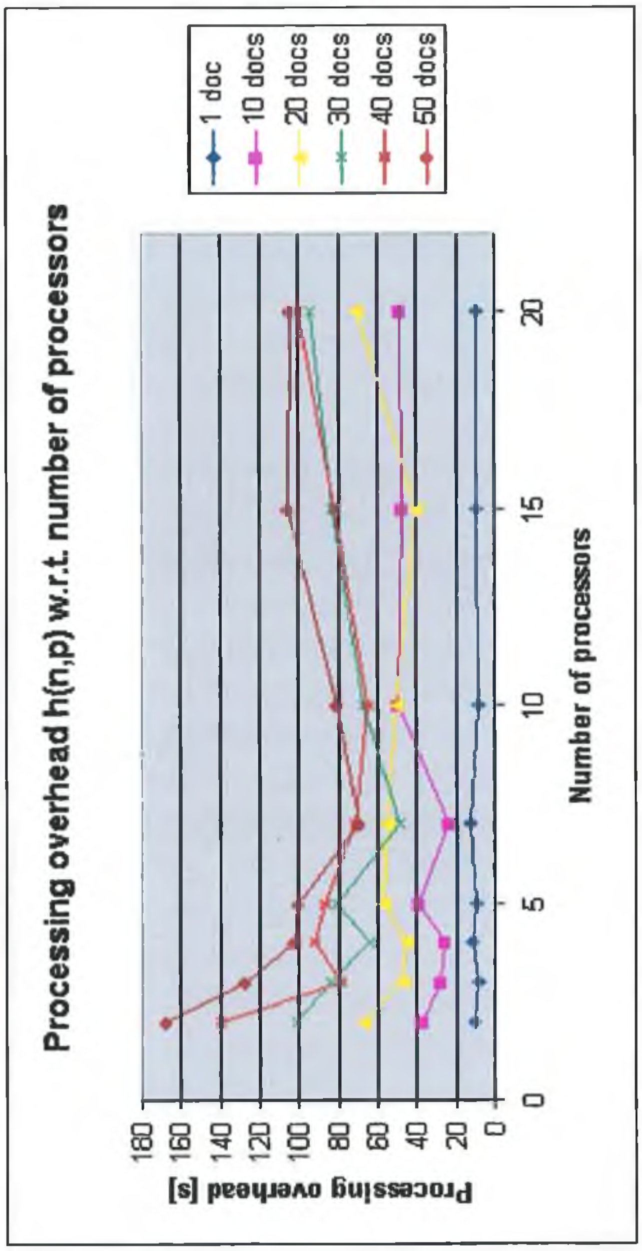


Fig. C.4 Processing overhead with respect to number of processors (Fig. 9.24)