DOCXS
A Distributed Computing Environment for Multimedia Data Processing

Tobias Lohe, Michael Fieseler, Steffen Wachenfeld and Xiaoyi Jiang
Department of Computer Science, University of Münster, Einsteinstraße 62, D-48149 Münster, Germany

Keywords: Distributed multimedia systems, workflow systems, visual programming.

Abstract: This paper presents DocXS, a distributed computing environment for multimedia data processing, which was developed at the University of Münster, Germany. DocXS is platform independent due to its implementation in Java, is freely available for non-commercial research, and can be installed on standard office computers. The main advantage of DocXS is that it does not require its users to care about code distribution or parallelization. Algorithms can be programmed using an Eclipse-based user interface and the resulting Matlab and Java operators can be visually connected to graphs representing complex data processing workflows. Experiments with DocXS show that it scales very well with only a small overhead.

1 INTRODUCTION

In this paper we present DocXS (Distributed Operator Construction and eXecution System), a computing environment for multimedia data processing. DocXS harnesses the power of distributed computing, allows the easy combination and integration of existing algorithms or software packages, and facilitates the scientific exchange among researchers. Additionally DocXS provides a visual programming environment for the definition of workflows based on smaller units called operators.

In the literature, several reports on distributed systems for multimedia data processing exist. One of the first reported systems is DIPE (Zikos et al., 1997), which uses binary executables as operators. DIPE provides no control structures like branches or loops and is the only system without a visual programming interface.

The LONI pipeline processing environment (Rex et al., 2003) also uses binary executables as operators and is able to distribute operators automatically, but does not provide any control structures.

Khoros/Cantata (Konstantinides and Rasure, 1994; Young et al., 1995) provides the control structures IF/ELSE, SWITCH, WHILE and COUNT, but the operators (also binary executables) have to be manually distributed by the user.

The IRMA (Image Retrieval in Medical Applications) platform (Güld et al., 2003) is able to automatically distribute operators, which have to be written in C++, but only provides an IF/ELSE control structure.

SCIRun (Parker et al., 1997) finally supports only C++ operators, provides no control structures and supports only manual distribution.

In contrast to DocXS, all these systems lack the possibility to include operators written in Matlab or Java and to combine operators from different languages in the same workflow. Also none of the systems supports a combination of loops and automatic distributed processing. DocXS in contrast allows branches as well as loops and automatically distributes operators. Further, to facilitate identical operations on multiple data, DocXS allows use of a construct called FOREACH. This loop-like construct is very useful as the identical operations are independent and can be automatically distributed and processed in parallel.

This paper is structured as follows. Section 2 gives a detailed overview about the architecture and implementation of DocXS. In Section 3 we present experimental results which include a performance analysis. The paper concludes with a discussion of our achievements in Section 4.

Lohe T., Fieseler M., Wachenfeld S. and Jiang X. (2007).
DOCXS - A Distributed Computing Environment for Multimedia Data Processing.
In *Proceedings of the Second International Conference on Signal Processing and Multimedia Applications*, pages 379-382
DOI: 10.5220/0002140003790382

2 ARCHITECTURE AND IMPLEMENTATION

We will use several technical terms to describe DocXS. An *operator* is some piece of code that executes arbitrary computations. A *chain* is a higher-order definition of a workflow consisting of several connected operators and control structures (like IF/ELSE, WHILE or FOREACH) which form a directed graph. An example of a simple chain which represents a process to detect edges in images is shown in Figure 1. It can be seen that operators can have multiple typed and labeled inputs and outputs, which are called *ports* in DocXS.

A chain which represents a specific algorithm can be applied by different users onto different data at the same time. Each application leads to an active instance of the chain within the system, which is called a *task*.

DocXS is designed to support Matlab and Java operators and allows to combine them in the same chain. It uses a lightweight API for the addition of new operators which makes the integration of already existing code into DocXS very easy. The chains, which can be constructed by combining operators and control structures, are designed to be able to model arbitrary workflows, which are automatically analyzed, distributed, and computed in parallel by the system. Furthermore, DocXS emphasizes the scientific collaboration inside a group or company, as it allows to share operators, chains, and data.

DocXS is implemented in Java and requires only a Java virtual machine to run. Therefore DocXS is completely platform independent. For the execution of Matlab operators of course a valid Matlab installation and license is required.

2.1 Distributed System Architecture

The architectural overview of the distributed DocXS system can be seen in Figure 2. The system consists of various components that can be distributed among different computers. The central server hosts the *Kernel*, which serves as the main coordinator and controller of the system. Tightly integrated with the Kernel is the server running the central database. The distributed execution of tasks is performed on multiple computers each running an *Executor*. The number of Executors is not limited.

DocXS provides two separate user interfaces: The so-called *SystemGUI* to create operators and chains and the *WebGUI* to execute chains without requiring programming knowledge. The WebGUI is implemented using the JavaServer Faces technology, runs

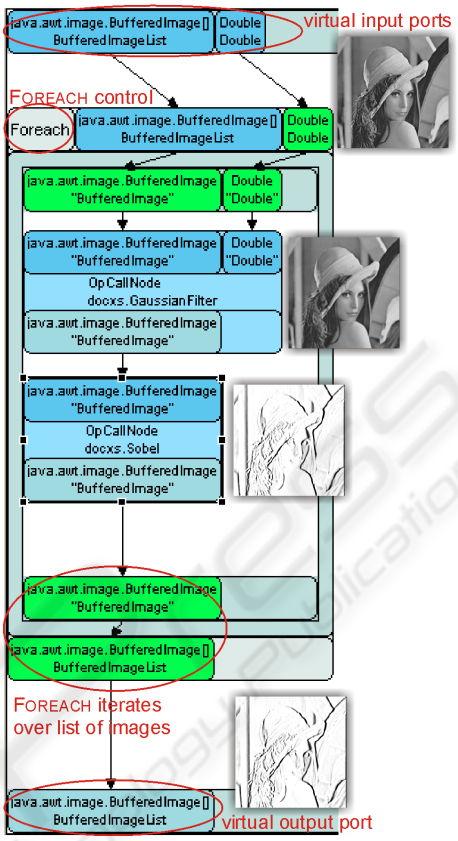

Figure 1: A chain representing an edge detection algorithm.

in an Apache Tomcat servlet container, and can be used with any modern Web browser. The SystemGUI of DocXS is based on the Eclipse Rich Client Platform (McAffer and Lemieux, 2005) and does not run on a server, but on the developers' computers.

2.2 Operators and Chains

For the creation of Java operators using the SystemGUI, the full functionality of the Eclipse Java IDE (syntax highlighting, code completion, refactoring support) can be employed, while for Matlab operators only syntax highlighting is provided. All built-in data types of Java and Matlab can be used as input and output parameters for operators.

Integrating existing Java code or creating new Java operators is done by simply implementing an interface and defining getter and setter methods. Matlab operators just need a `main` function which can be called by DocXS. A single DocXS operator may consist of several Java classes or Matlab files.

Available operators can be inserted into a chain using drag-and-drop. Java and Matlab operators can be mixed in an arbitrary manner inside a chain. The

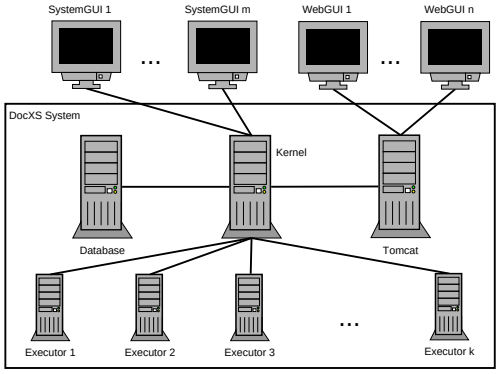

Figure 2: Distributed system architecture of DocXS.

data flow is represented by edges between ports. Necessary type conversions are done automatically when the chain is executed.

For the definition of complex workflows several control structures are provided. Conditional execution can be expressed using the IF/ELSE or SWITCH control, and loops using the WHILE control. Especially important is the FOREACH control structure that allows a user to execute a part of the chain for every element of a list or array. As the identical operations applied to each element are independent of each other, the FOREACH can be automatically distributed among the Executors.

2.3 Task Execution

Available chains can be executed using the WebGUI. After the user has selected the required input parameters, the execution of the task can be started. The Kernel analyzes the task and splits it into several parallel *jobs* for distribution. An internal scheduler assigns the resulting jobs to the available Executors, where they are executed in parallel. The Kernel also takes care about handling the dependencies between jobs of the same task and the coordination of the Executors running the jobs.

The Executor analyzes the job, provides and converts the input data, executes the contained operators using the Java Reflection API or the JMatLink Java-Matlab connector, takes care about the proper execution of control structures and writes the output data.

2.4 Data Storage

The system data—operators, chains, tasks, task parameters, and task results—is stored in a central database. Images and other media files are stored using the file system and only links to their location are stored in the database. We use Hibernate (Bauer and King, 2006) as object-relational mapper, which delivers a convenient object-oriented abstraction layer of the underlying relational SQL database. Therefore almost any relational database system can be used with DocXS and a switch from one database system to another is possible without changing any line of code and requires only to change the according system properties. We currently use the IBM DB2 Express-C database system.

3 EXPERIMENTAL RESULTS

In this section we present some experimental results considering the performance of DocXS. We use a cluster of k standard office computers as Executors, each having a 1.7 GHz Intel Pentium 4 CPU and 512 MB RAM, and a non-dedicated server with two 2.8 GHz Intel Xeon Dual-Core CPUs and 6 GB RAM for the Kernel. The database runs on a non-dedicated server with an AMD Athlon XP 2000 CPU and 512 MB RAM. All computers are connected using a 100 MBit Ethernet network. We used repeated test runs and took the median of all runs to reduce the impact of resulting outliers.

3.1 Estimation of System Overhead

To estimate the computational overhead of DocXS for system management and task distribution, we use a task that consists of a NOP operator implemented in Java, which does nothing and simply returns the inputs without modification. The operator is placed in a FOREACH control so that the operator has to be executed for each input item. We measure the time DocXS needs to run such a task.

We show two different cases. In the first case (*NOP-few*) the input data consists of 64 integer values to keep the time for data distribution to a minimum. The second case (*NOP-large*) involves a larger amount of data, a set of 64 files (each 1.3 MB), that has to be distributed. This case not only reflects the network speed, but moreover the internal handling of the data by the system.

Table 1 shows the total time needed for both cases depending on the number k of participating Executors and the execution time of the same tasks without DocXS. It can be seen that DocXS itself causes only a small overhead. The overhead in the *NOP-large* case decreases with higher numbers of Executors due to distributed I/O. Both cases show that using DocXS already pays off if a task takes about a minute without DocXS, in the case of low I/O demands even less.

Table 1: Execution times for different numbers k in comparison to the execution time of the task without DocXS.

k	*NOP-few*	*NOP-large*	*Comp-few*	(speedup)	*Comp-large*	(speedup)
No DocXS	$<$ 1ms	1m 02s 407ms	59m 52s	($=$ 1.00)	1h 03m 59s	($=$ 1.00)
1	875 ms	1m 57s 801ms	1h 05m 06s	($\times$ 0.92)	1h 06m 01s	($\times$ 0.97)
4	6s 546ms	1m 08s 675ms	16m 34s	($\times$ 3.61)	17m 12s	($\times$ 3.72)
8	8s 140ms	1m 17s 640ms	8m 22s	($\times$ 7.16)	9m 15s	($\times$ 6.91)
16	13s 191ms	1m 01s 935ms	4m 14s	($\times$14.17)	5m 02s	($\times$12.72)

3.2 Performance Comparison

To measure the performance of our system we use two cases very similar to the cases for the overhead estimation. Both cases use a computationally intensive Java operator. While the first case (*Comp-few*) uses only primitive data types, in the second case (*Comp-large*) the amount of data which has to be transferred over the network and into memory is higher. For both cases the speedup of DocXS in comparison to a single computer without DocXS, calculated as speedup $= T_{\text{no DocXS}}/T_{\text{DocXS}}$, is shown.

It can be seen in Table 1 that DocXS scales very well in the *Comp-few* case. For one Executor ($k = 1$) DocXS needs slightly longer due to the already discussed overhead. But the speedup grows linearly with an increasing number of Executors and for $k = 16$ the task is finished more than 14 times faster than on a single computer. In the *Comp-large* case, which involves sending larger amounts of data over the network, DocXS scales very well, too. Tasks can be finished almost 13 times faster using DocXS instead of a single computer.

DocXS can also make efficient use of a multiprocessor computer by running an Executor instance on each processor available in the system. Tests using a single multiprocessor computer with eight CPUs resulted in a speedup of 7.73 (*Comp-few*) resp. 6.53 (*Comp-large*).

4 CONCLUSION

We presented DocXS, a distributed computing environment for multimedia data processing. The main advantage of DocXS is that it does not require its users to care about code distribution or parallelization, but handles these issues automatically. Algorithms can be programmed using an Eclipse-based user interface and the resulting Matlab and Java operators can be visually connected to a complex workflow using various branch and loop control structures. Additionally the scientific exchange of operators, algorithms, and data is facilitated using a central database and two user interfaces, one for developers and one for system users.

We showed that DocXS produces only a small overhead and that it scales very well for computationally expensive tasks. As DocXS is going to be freely available for non-commercial research and may run on cheap PC hardware, it is a useful tool which can simplify and facilitate every researcher's work.

REFERENCES

Bauer, C. and King, G. (2006). *Java Persistence with Hibernate*. Manning.

Güld, M. O., Thies, C., Fischer, B., Keysers, D., Wein, B. B., and Lehmann, T. M. (2003). A platform for distributed image processing and image retrieval. In *Visual Communications and Image Processing 2003*, volume 5150 of *Proceedings of SPIE*, pages 1109–1120.

Konstantinides, K. and Rasure, J. R. (1994). The Khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing*, 3(3):243–252.

McAffer, J. and Lemieux, J.-M. (2005). *Eclipse Rich Client Platform: Designing, Coding, and Packaging Java Applications*. Addison-Wesley Professional.

Parker, S., Beazley, D., and Johnson, C. (1997). Computational steering software systems and strategies. *IEEE Computational Science and Engineering*, 4(4):50–59.

Rex, D. E., Ma, J. Q., and Toga, A. W. (2003). The LONI pipeline processing environment. *NeuroImage*, 19(3):1033–1048.

Young, M., Argiro, D., and Kubica, S. (1995). Cantata: Visual programming environment for the Khoros system. *Computer Graphics*, 29(2):22–24.

Zikos, M., Kaldoudi, E., and Orphanoudakis, S. C. (1997). DIPE: A distributed environment for medical image processing. In *Proceedings of MIE'97 (Medical Informatics Europe)*, pages 465–469.