Vol. 12, No. 3, Sept. 2003, pp. 205-323

International Journal of Cooperative Information Systems
© World Scientific Publishing Company

INTRA-SUBNET LOAD BALANCING IN DISTRIBUTED WORKFLOW MANAGEMENT SYSTEMS

THOMAS BAUER
Dept. RIC/ED, DaimlerChrysler Research and Technology
thomas.tb.bauer@daimlerchrysler.com
P.O. Box 2360, 89013 Ulm, Germany

MANFRED REICHERT, PETER DADAM
Dept. Databases and Information Systems, University of Ulm
{reichert, dadam}@informatik.uni-ulm.de
Oberer Eselsberg, 89069 Ulm, Germany

Received (to be inserted
Revised by Publisher)

For enterprise-wide and cross-organizational process-oriented applications, the execution of workflows (WF) may generate a very high load. This load may affect WF servers as well as the underlying communication network. To improve system scalability, several approaches for distributed WF management have been proposed in the literature. They have in common that different partitions of a WF instance graph may be controlled by different WF servers from different subnets. The control over a particular WF instance, therefore, may be transferred from one WF server to another during run-time if this helps to reduce the overall communication load. Thus far, such distributed approaches assume that exactly one WF server resides in each subnet. A single server per subnet, however, may become overloaded. In this paper, we present and verify a novel approach for replicating WF servers in a distributed workflow management system. It enables an arbitrary and changeable distribution of the load to the WF servers of the same subnet, without requiring additional communication.

Keywords: workflow management, scalability, distributed workflow execution, load balancing, hashing.

1. Introduction

Workflow management systems (WfMS) deliver a state-aware control service for process-oriented applications [1,2]. Designed for a distributed environment, they increase the number of work processes that can pass through an electronic workplace. In detail, the WfMS routes, assigns, activates, and tracks the tasks of workflow (WF) instances according to their predefined WF schema.

This work was partially performed within the research project "Scalability in Adaptive Workflow Management Systems", which was funded by the Deutsche Forschungsgemeinschaft (DFG).

The key advantage of WfMS is that they make WF-based applications easier to develop and to maintain when compared to traditional coding systems [3]. By extracting business process logic from the code of the application programs, one obtain a set of activities, which represent the application components, instead of one big and monolithic program. The process logic between these activities is specified in a separate control and data flow definition. It determines the order and the conditions for the execution of these activities as well as the data exchanged between them. Usually, workflows can be modeled at a high semantic level, which decreases error rates and increases process consistency. In addition, non-automatic activities can be associated with user roles, which enables the WfMS to insert corresponding tasks into the worklists of potential actors at run-time.

A serious deficiency of existing WfMS is their insufficient scalability in case of a high number of users and WF instances [4]. In such a scenario, the load of the WF servers and of the underlying communication network may become very large [5,6]. To avoid overload situations, therefore, the load has to be distributed to multiple system components in an appropriate manner. In this paper, we focus on approaches for avoiding overload situations in a distributed WfMS (cf. Fig. 1); i.e., a WfMS with WF schema partitioning and distributed control of the different partitions for the respective WF instances [7,8]. This is achieved in a way that does not imply any significant disadvantages for the communication behavior; i.e., additional communication is avoided as far as possible.

1.1. *Workflow Partitoning and Distributed Workflow Execution*

First of all, we summarize important properties of the ADEPT WF distribution model [9], which are necessary for the understanding of this paper. We consider an enterprise-wide or even cross-organizational application scenario, where business processes often span multiple domains of an organization and where the large number of users and concurrently active WF instances may result in a very high load [4,10,11]. This load is caused by the numerous tasks the WfMS has to fulfill, like the refreshing of user worklists, the synchronization of workitem choices, the starting of activity programs, or the transferring of activity parameter data.

In order to avoid an overloading of system components (WF servers, subnets, gateways), in our ADEPT approach [12,13], a particular WF instance may not always be controlled by only one WF server. Instead, several disjoint partitions of the control flow graph – called WF graph for short – may be created at build-time and the corresponding WF instances may then be controlled "piecewise" by different WF servers during run-time [7]. An example is given in Fig. 1, where a WF graph is subdivided into three partitions controlled by the WF servers s_1, s_2, and s_3. If, for example, activity a is completed (cf. Fig. 1 a), the control over partition 3 will be transferred to the WF server s_3, whereas the WF server s_1 will retain control over the partition 1 (cf. Fig. 1 b). To perform such a *WF instance migration*, a description of the state of the WF instance has to be transferred from the source to the target server. In order to reduce the synchronization effort between the different WF servers, activities of parallel execution branches (e.g., activities c and d in Fig. 1) are controlled independently from each other. For example, the server s_2 does not need to know the state of activity d, which is controlled by WF server s_3 .

Why do we need WF graph partitioning and WF instance migrations at all? A very important goal of our work is to reduce the overall communication costs. Experiences we made with existing WfMS have shown that numerous messages between a WF server and its clients have to be transferred; e.g., when updating user worklists or when transferring activity parameter data [14]. In some cases, large data volumes must be exchanged, which may result in an overloaded communication sys-

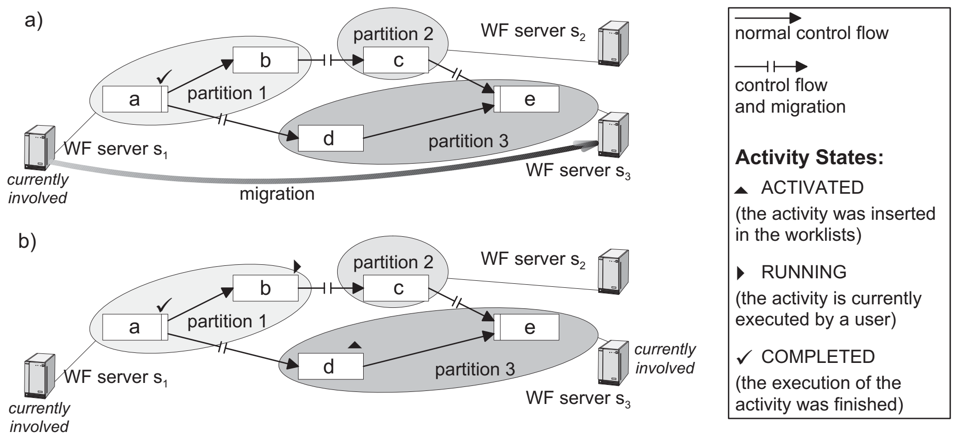

Figure 1: **a)** Migration of a WF instance from WF server s_1 to s_3 **b)** Resulting state of the WF instance after this migration: Activity d is activated and both servers, s_1 and s_3, are now involved in the control of the WF instance. Additionally, activity b has been started.

tem. To avoid such an overloading, in ADEPT, for each activity the WF server is chosen in a way that minimizes the overall communication effort at run-time. This is achieved by calculating the minimum of a cost function [7], which also considers the data flow between activities and the costs for WF instance migrations. As a rule of thumb, in most cases the WF server of an activity is selected in that domain, in which the majority of its potential actors resides (a domain is considered as a subnet together with the WF servers and clients residing in it). In doing so, consecutive activities with the same server assignment are combined to partitions (i.e., compound sub-graphs of the WF graph). To avoid misunderstandings, it is important to mention that in ADEPT the unit of distribution is the partition and not the single activity. Usually, the latter granularity is too low to minimize the overall communcitation costs, since migrations themselves require some communication between WF servers. How to calculate an optimal partitioning for a given WF graph, however, is outside the scope of this paper and was reported in earlier publications of our group [7,9]. In any case, the concrete number of partitions and their composition depend on the respective WF type and its attributes.

WF schema partitioning and WF instance migrations offer several advantages:

- The total communication costs can be significantly reduced since subnet-spanning communication between a WF server and its clients can be avoided in many cases.
- Response times are improved and system availability is increased since the execution of activities does not require a gateway or a WAN (Wide Area Network).

Comparable observations have been made by several other research groups [8], which consider issues related to distributed WF management as very important as well. Since business processes very often span multiple domains of an organization, for many cases, WF graph partitioning and WF instance migrations lead to significantly better results than distributing whole workflows at the instance level (see also Section 2).

Following the sketched approach, the partitions of a WF graph are statically defined; i.e., the WF servers have to be assigned to the WF activities already at build-time. In some cases, however, this static approach is not sufficient to achieve good results, especially if *dependent actor assignments* are used (e.g., "activity n has to be performed by the same actor as the preceding activity m"). As the potential actors of such a dependent activity can only be calculated at run-time, it is advantageous to determine its WF server at run-time as well. The WF server can then be chosen in a domain that is beneficial for these actors.

In ADEPT, we use so-called *variable server assignments* [6,15] to satisfy this requirement. As an example, such an assignment may require that activity n has to be controlled by the server from that domain to which the actor of the preceding activity m belongs. Variable server assignments are defined at build-time and are evaluated at run-time in order to calculate the server that has to control the dependent activity. Generally, the use of variable server assignments results in significantly reduced communication costs [5].

1.2. *Problem Statement and Contribution of the Paper*

As already mentioned, the most important goal of the ADEPT approach is to distribute WF activities to WF servers in a way that minimizes the overall communication costs during run-time. For each WF activity the optimal subnet for its control is calculated automatically at build-time. Corresponding to this, the activities are assigned to the WF servers by *server assignment expressions*. Thereby, consecutive activities with the same server assignment build one partition. Generally, the problem with such a commonly used approach is that only one WF server can be used for a given domain (cf. Fig. 1). Otherwise, it would be ambiguous which server has to control an activity assigned to this domain.

In order to avoid WF server overload, we have to overcome this restriction. As depicted in Fig. 2, it should be possible that a domain comprises more than one WF server. Since these WF servers are located in the same subnet, they are equivalent with respect to the communication costs occuring during activity execution. It does not matter, therefore, which of them actually executes a particular activity. Consequently, it is not possible to calculate a “most suited” WF server with respect to a particular domain at build-time. This means that only the domain of the WF server controlling an activity is determined at build-time, but not the WF server itself. We therefore require a method that dynamically selects a concrete WF server for the execution of an activity at run-time (e.g., in Fig. 2 for activity c of partition 2 one of the servers $s_{2,1}$, $s_{2,2}$, or $s_{2,3}$ has to be chosen). Following this approach, one has to realize some kind of “load balancing” between these servers. In addition, it must not require any communication overhead. For example, it should not be necessary to synchronize the WF servers of the same domain or to migrate WF instances between them. Such an overhead cannot be accepted in a heavily burdened WfMS.

In this paper, for the first time, we examine how multiple WF servers may be realized within the same subnet in an intelligent way; i.e., without producing the overhead mentioned above. The corresponding method has to respect some WF specific peculiarities, which are discussed in Section 3.3. In addition, the following requirements have to be fulfilled:

- **Requirement 1:** The WfMS administrator must be able to distribute the load to an arbitrary number of WF servers within one domain. This load distribution must be realizable in an arbitrary and definable ratio that corresponds to the capacity of these WF servers. It must be possible, for example, to distribute the load among two existing WF servers of a domain at the ratio of 3:2. This is the only way to ensure that no WF server will become overloaded if an arbitrary set of server computers is assumed.

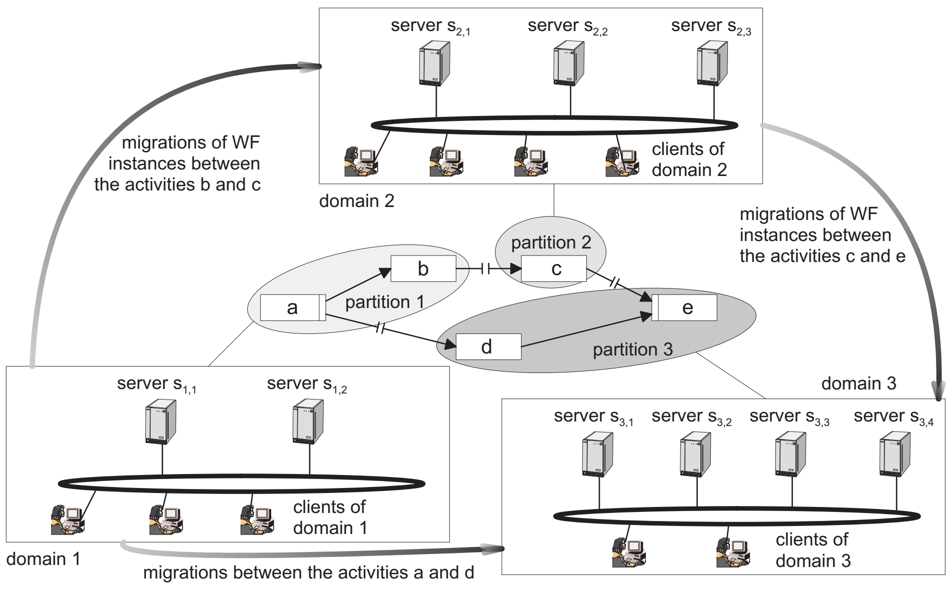

Figure 2: Using several WF servers within the same domain (with assignment of partitions to domains, but not to a concrete WF server).

- **Requirement 2:** The usage of several WF servers within one domain must not affect the advantages achieved by the distribution model of ADEPT (e.g., the reduction of the communication load). During WF execution, therefore, it should not require any (additional) communication between WF servers of the same domain. The selection of a concrete WF server for an activity instance must be possible without requiring synchronization between these WF servers.

- **Requirement 3:** It is not favorable to use a central server for distributing the load, since this would cause additional communication costs. In addition, such a central component would represent a (potential) bottleneck and affect the availability of the WfMS. We, therefore, need a distributed approach for selecting the server of a given activity instance.

- **Requirement 4:** To be applicable in a broader context, the approach has to work in conjunction with variable server assignments (cf. Section 1.1) as well. In addition, in the context of dynamic WF instance modifications (e.g., to insert or delete activities during run-time [12,13]), it must be possible to detect the server that actually controls a given WF instance without large effort [16].

To summarize, our goal is to develop an approach that allows an arbitrary and changeable distribution of the load to the WF servers and that does not require any additional communication.

1.3. *Assumptions and Preconditions*

This section discusses issues with respect to the system environment assumed by this paper. Considering the operative usage of WfMS the following assumptions hold:

- The WF servers control a large number of WF instances and, therefore, have to cope with a high load, especially if enterprise-wide and cross-organizational applications have to be considered. In addition, in such an environment, it can be assumed that powerful computers are used as WF servers. Each single server machine may therefore control a large number of WF instances.
- The computers used as WF servers are (to a certain degree) reserved for this purpose. There is no reason to assume that their capacity fluctuates due to other duties.
- The capacity of the WF servers of a domain is somewhat larger than required (i.e., there is a reserve) since the total load of the domain may fluctuate. Therefore, smaller fluctuations in the load of a single server can be compensated as well.

The next section discusses some solution approaches generally possible. In Section 3 we describe how the concrete server of an activity instance is selected in our approach. Section 4 considers important issues concerning the modification of the chosen load distribution. The approach is verified in Section 5 by means of simulations. In Section 6 related approaches are discussed. The paper concludes with a summary and an outlook on future work.

2. General Solution Approaches

In this section we discuss general approaches that may contribute to cope with a high WF server load (cf. Fig. 3). Besides different methods for the replication of WF servers, we discuss several other approaches for improving WfMS scalability, like the use of a powerful, centralized computer system, load balancing at the WF instance level, and load balancing by changing server assignments.

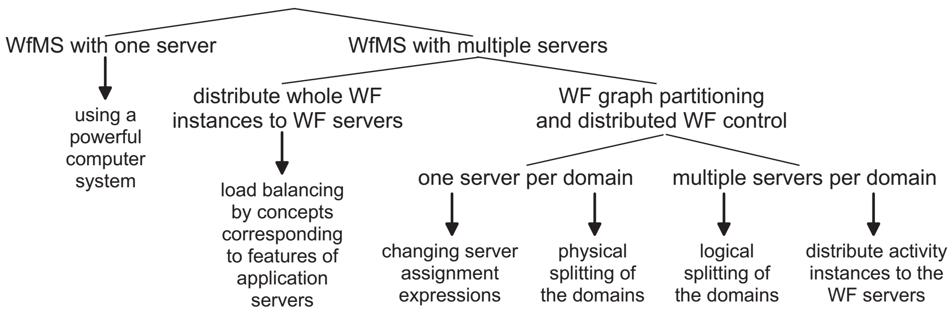
WfMS with one server
WfMS with multiple servers
using a powerful computer system
distribute whole WF instances to WF servers
WF graph partitioning and distributed WF control
load balancing by concepts corresponding to features of application servers
one server per domain
multiple servers per domain
changing server assignment expressions
physical splitting of the domains
logical splitting of the domains
distribute activity instances to the WF servers

Figure 3: General Approaches for Coping With High WF Server Loads

An obvious possibility is to use a single, but extremely powerful computer (e.g., a centralized mainframe system). For WfMS with one WF server, the ADEPT model could be applied without any modification. High performance systems, however, normally come along with very high acquisition costs, and even such systems may reach their limits. Apart from this, distribution issues often stem from organizational facts, like business processes spanning multiple domains of an organization.

As pointed out by several research groups [17,7,18,8,11,19]), therefore, it is worth to deal with issues related to WfMS with multiple WF servers.

For WfMS with multiple WF servers, a general possibility would be to distribute whole WF instances to the WF servers [17,18]; i.e., to realize load-balancing at the WF instance level. This can be achieved, for example, by using the load balancing capabilities of application servers. Obviously, this approach contributes to avoid the overloading of WF servers. As explained in the previous section, however, it may result in an unfavorable communication behaviour and in increased response times, if the potential actors of a WF are geographically spread.

To avoid these problems, several approaches for distributed WF management have been proposed in the WF literature [20,21,8,22]. As ADEPT, most of them use WF graph partitioning and distributed WF control. The different proposals made in this context are discussed in more detail in Section 6.2. For the remainder of this section, we present general possibilities for coping with high WF server loads in a distributed WfMS (as the one described in Section 1.1).

In a WfMS with multiple WF servers and with WF graph partitioning, the load of a WF server may be reduced by re-assigning some of the activities under its control to WF servers of other domains. This can be achieved, for example, by changing the server assignment expressions of these activities accordingly. Corresponding changes may be applied statically (and may therefore be only valid for future activity instances) as well as dynamically during run-time. First of all, following this approach, there is no need for WF server replication within one domain. This method, however, increases the overall communication costs, since the original servers have been chosen in a way that minimizes these costs. It, therefore, violates the basic idea of the ADEPT model. In addition, it cannot be used if the WF servers are "globally overloaded"; i.e., if the total load is higher than the current capacity of the WF servers. Note that with this approach it is not possible to introduce additional WF servers in order to increase the total capacity of the WfMS since only one WF server per subnet is used.

In summary, the approaches discussed so far (powerful centralized computer system, load balancing at the WF instance level, and change of server assignments) are not suited to completely satisfy our requirements. In the following we discuss more sophisticated solutions, which are based on the replication of WF servers.

2.1. *Splitting Domains*

If a WF server gets overloaded, the corresponding domain may be divided into several sub-domains, each of them possessing its own server and subnet. Following this approach, additional servers and subnets are introduced. As a variant, one may create only logical domains, each of them with its own server, but belonging to the same subnet. Independent from which variant is applied, the users of the "original domain" have to be distributed to the resulting domains. This approach is recommendable if a "natural splitting" of the domain exists; i.e., the domain can be separated into largely disjoint parts (with separate activities and separate users). If, in addition, the corresponding subnet is heavily burdened, a splitting of the domain into disjoint subnets will be recommendable as well. For this case, however, the splitting of the domains requires the rebuilding of the communication network.

Both variants have some serious disadvantages: In many cases, a proper splitting of the domains is not possible. As a consequence, there is no reasonable distribution of the users to the new domains. In addition, it is difficult to distribute the load according to the demanded ratio. The load of a WF server depends on the users as well as on the number and the kind of activities which are assigned to its domain. Thus, it is not possible to directly define a ratio for the distribution of the load to the WF servers. This violates Requirement 1 of Section 1.2.

2.2. *Using Multiple Workflow Servers Within One Domain*

If multiple WF servers are used within one domain (cf. Fig. 2), the domains will not have to be changed with respect to users and subnets. Instead, to cope with overload situations, additional WF servers may be introduced into the respective domain. The problem then is to find a way to distribute activity instances to multiple servers of one domain. This will be discussed in Section 3.

With this approach, instead of a concrete server for each activity, only the corresponding domain has to be defined at build-time. The assignment of a concrete server to an activity is considered as a physical aspect that has not to be handled by the WF designer at build-time. The usage of several WF servers per domain, in principle, enables an arbitrary distribution of the load to them. A positive side effect is the increase of system availability, since users are connected to several WF servers of their domain. If one of them breaks down, they may continue their work with the other ones.

The usage of multiple WF servers within one domain may influence the communication behavior when updating user worklists. Since it increases or decreases the communicated data volume only marginally, depending on the method used for updating the worklists, we do not discuss this aspect further in this paper. A detailed discussion of this topic can be found in [9].

To sum up, splitting of domains is not appropriate to solve the given problem. Instead, it is more favorable to allow the usage of multiple servers per domain. Until now, we have not inspected ways to select the concrete WF server of a given domain, which has to control a particular activity instance. This is crucial for the presented approach. We discuss related issues in the next section. Thereby, the goal is to achieve an arbitrary definable distribution of the load to the WF servers.

3. Selecting the WF Server of a Domain

In the previous section, we have shown that it is favorable to replicate WF servers within a domain (cf. Fig. 2). Following this approach, an activity server assignment no longer defines a concrete WF server. Instead, it only fixes the domain to which the WF server (for the control of the respective activity) should belong.

The problem now is to find a way to select a concrete server of the designated domain for a given activity instance. Such a selection method has to fulfill the requirements as discussed in Section 1.2. Otherwise it would counteract the advantages of any efficient distribution model (such as ADEPT). In particular, an arbitrary distribution of the load must be possible in order to enable the WfMS administrator to define a suitable configuration with respect to the number of WF servers in use and the corresponding load distribution. This task could also be performed automatically by a monitoring component; if a WF server has to cope with a load that is too high or too low in the long term, this component may redistribute the load or may even start and stop WF servers. In addition, any solution approach also has to consider more advanced system features (such as variable server assignments) and, as far as possible, it should be realized without additional communication.

Issues related to the changeability of the load distribution during run-time (Requirement 5) are presented in Section 4. In the following, we discuss several solution approaches: static, load dependent, random, and hash server assignments. Finally, a method is developed that fulfills the Requirements 1-4.

3.1. *Static Assignment of Workflow Servers at Build-Time*

A simple approach to distribute the load between the WF servers of a domain is as follows: Just as with the domain, the WF server which has to control a particular WF activity may be explicitly defined at build-time in the WF model (e.g., in Fig. 2

" instances of activity c have to be controlled by the server $s_{2,3}$ of the domain 2"). Using this approach, distinct activities, which belong to distinct partitions of the WF graph, may be controlled by different WF servers of the same domain.

Following this approach, the servers for the WF activities are statically defined (cf. Section 2.1). Consequently, it is almost impossible to realize the required load distribution. This, in turn, violates Requirement 1. In addition, this static approach cannot be used in conjunction with variable server assignments. Such an assignment is defined by an expression that is evaluated at run-time in order to determine the domain of the activity (e.g., "server in the domain of the actor of activity a"). Using these expressions, it is not possible to define the WF server within the resulting domain explicitly.

3.2. *Load-Dependent Selection of Workflow Servers at Run-Time*

A desired distribution of the total load between the WF servers of a domain can be achieved by choosing the server with the lowest current load (compared to its intended load) for activity execution. To identify this server, for each domain the corresponding load information must be available. Generally, there are two possibilities to offer this information:

1. The load information may be distributed to all servers of the WfMS – periodically or by "piggybacking" [23] together with other communications. When performing a WF instance migration, this information will be locally available within each domain. The target server of the migration, therefore, can be directly selected.

2. The load information may be only notified to a dispatcher in the local domain. Consequently, this dispatcher performs the selection of a WF server if an activity instance has to be executed.

The first approach results in a larger effort with respect to the distribution of the load information – each WF server has to be informed about the current load of all other servers. The second approach reduces this effort. Only the dispatcher, which is located in the same domain, requires the corresponding load information of a WF server. With this approach, however, for an activity instance the selection of a WF server becomes more expensive since a communication with the dispatcher of the respective target domain is required. In addition, the dispatcher is a central component of the domain, which is required for each migration. It, therefore, reduces the availability of its domain. To sum it up, both approaches violate Requirement 2. The second approach additionally violates Requirement 3.

The main advantage of load-dependent methods is their capability to balance fluctuations in the load of the WF servers. Such fluctuations may occur as a result of WfMS external events, or when some of the WF instances controlled by a server temporarily generate a particularly high load. The key disadvantage of both approaches is the additional effort required for communication and synchronization. Therefore, the load the WF servers are able to manage does not grow proportionally to their capacity.

It is worth mentioning that similar algorithms have been used for process scheduling [24,25] in distributed operating systems. In this area, however, the situation is somewhat different. In advance, almost no information is available about the resources required by an operating system process. For this reason, normally, dynamic methods are used for process scheduling. As opposed to this, in WfMS much more information about WF types as well as WF participants is available at build-time. Nevertheless, the basic problems, as known from process scheduling methods, occur in our scenario as well. In particular, the success of these load-balancing methods heavily depends on the quality of the available load information as well as on an adequately chosen frequency for the load information exchange. For process scheduling methods, generally, it is very difficult to guarantee a good quality of these values [25]. In WfMS the situation is even worse since workflows may have a complex internal structure and may stay at the selected server for several weeks. During this time, the load situation of the respective WF server may totally change. To summarize, on the one hand load-dependent methods are an (academically) interesting approach since they offer the possibility to compensate load fluctuations. On the other hand, they generate additional communication load and it is difficult to predict their behavior. We discuss the applicability of process scheduling methods for WfMS in more detail in Section 6.1.

3.3. *Random Selection of Workflow Servers at Run-Time*

When a new partition is entered during the execution of a WF instance (i.e., when starting or migrating the WF instance), the WF server of the target domain may be chosen randomly.

For this purpose, a random number z of the interval $[0, 1)$ is calculated. This interval had been decomposed into disjoint sub-intervals, which were assigned to the WF servers of this domain. For the target partition under consideration, now, that WF server is selected for activity execution, in whose sub-interval z resides. Obviously, the probability that a specific WF server is selected corresponds to the length of its interval. Consequently, the same applies to its portion of the total load. The intervals, and therefore the distribution function, can be easily defined by the WfMS administrator according to the intended load of the server machines.

This random selection approach has several advantages: It allows an arbitrary distribution of the load, which can be easily modified by changing the sub-intervals. In addition, the approach can be used in conjunction with variable server assignments as well (cf. Section 1.1). It is even possible to distribute the load with respect to instances of a particular activity type to several WF servers. In case of a large number of WF instances, significant deviations from the intended load distribution, occurring due to shortcomings of the random number generator, are very improbable.

To randomly select a WF server, however, has one crucial disadvantage: Problems occur when parallel WF execution branches have to be joined. This is explained using the WF graph shown in Fig. 4. Assume that the migration $M_{b,c}$ of a corresponding WF instance will be performed previous to the migration $M_{d,e}$ of the same instance. Then, the concrete WF server within domain 3 will be selected when executing $M_{b,c}$. Since this server is normally not known to the WF servers of domain 2, it may happen that a different WF server from domain 3 is randomly chosen when the migration $M_{d,e}$ takes place. This will generate a problem when joining the parallel branches at activity f. If the different source servers of the migrations have to coordinate the target servers, additional communication is required. This violates Requirement 2.

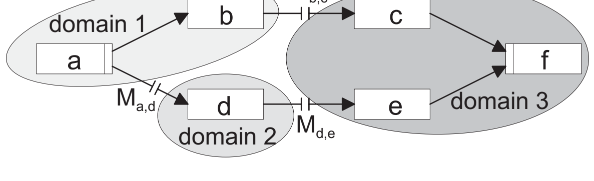

Figure 4: Problem with random server selections, when joining parallel branches.

3.4. *Hashing of Workflow Instances to Servers*

We now want to construct a selection method that keeps the positive properties of the previous approach, but tries to avoid the problems arising in conjunction with parallel joins. This succeeds with the following idea: The WF server is not chosen randomly as described in Section 3.3. Instead, for a particular WF instance, always the same WF server is selected within a given domain. This is achieved by using a hash function, which maps a WF instance specific date (e.g., the WF instance ID) to a number $z \in [0, 1)$. To realize this hash function, for example, a pseudo-random number generator may be used. Following this approach, for a given WF instance always the same number z and, therefore, the same WF server are calculated. With respect to the selection of the hash function, however, it is important that it produces a good distribution. Even for similar input data, it must be possible that they are mapped to widely spread random numbers.

By applying this hashing, all advantages of the approach discussed in Section 3.3 are preserved: An arbitrary distribution of the total load to the WF servers of a domain is possible, all concepts of ADEPT (incl. variable server assignments) can be further supported, and no additional communication is required. When joining parallel branches, no problem occurs since all activities belonging to the same partition of a WF instance are assigned to the same server. In summary, this approach fulfills the Requirements 1-4 of Section 1.2. It can be expected that it works better than load-dependent methods because no additional communication becomes necessary and the system behavior is more predictable. Since the WF servers possess a capacity reserve, a slightly unequal distribution of the load does not matter. This is another reason why we do not need a method for actively compensating such an unequal distribution.

Using the hash approach, the subset of the WF instances controlled by a particular WF server corresponds to its intended part of the load. Usually, each WF server controls a large number of WF instances. It, therefore, manages its intended load. Due to the large number of WF instances, potential differences in the effort necessary to control specific WF instances are balanced. Since the different servers of a domain control disjoint WF instances, no problems with data inconsistencies will occur (the static schema data is replicated at all servers).

For all these reasons, for the remainder of this paper, the hash approach is considered. It possesses the following properties which are essential for load balancing and for efficient WF execution:

- The concrete WF server that controls a WF activity is always determined at run-time.
- For different WF instances (of the same WF type), the same partition (e.g., partition 3 in Fig. 2) may be controlled by different WF servers.
- For a given WF instance, all activities assigned to the same domain (e.g., activities c, e, and f in Fig. 4) are always controlled by exactly the same WF server of this domain. Otherwise, synchronization overhead would become necessary between the WF servers of a domain.

In the following subsection, we show how to realize the distribution function and how to offer the required information to the WF servers. Section 4 discusses ways to additionally fulfill Requirement 5 (changeability of the load distribution).

3.4.1. *Realization of the Distribution Function*

Our approach uses a specific date of the WF instance *Inst* in order to calculate the WF server of the target domain D (e.g., when a migration has to be performed). For this, the domain specific distribution function $g^D(\mathit{Inst})$ is applied (cf. Algorithm 1):

The (unchangeable) instance specific date *InstDat* (e.g., the WF-ID) is hashed to a value z of the interval $[0, 1)$ which is then mapped to a server s_i of the domain D by the use of a function $f^D(z)$. Depending on the choice of this function, the load distribution to the WF servers can be controlled. Since a hash function always calculates the same result z with respect to a given input value *InstDat*, the WF instance *Inst* is always controlled by the same WF server s_i with respect to domain D.

Algorithm 1 (Calculation of the Distribution Function $g^D(Inst)$).

```
input
    Inst:   WF instance for which the WF server has to be calculated
    D:   domain the WF server belongs to
result
    logical ID of the WF server (of the domain D) selected for the WF instance Inst
begin
    $InstDat = instance\_specific\_data(Inst)$;
    $z = hash(InstDat)$;
    function $f^D(z)$ defined by $a_1^D \dots a_{n-1}^D$ and $s_1 \dots s_n$:
        case $z \in [0, a_1^D)$ : return $s_1$;
        case $z \in [a_1^D, a_2^D)$ : return $s_2$;
        ...
        case $z \in [a_{n-1}^D, 1)$ : return $s_n$;
end.
```

The probability that server s_i is selected corresponds to the length of its assigned interval $[a_{i-1}, a_i)$, assuming that the values of z are equally distributed within $[0, 1)$. This can be achieved by using a "good" hash function. Algorithm 1 starts with a WF instance specific value. For this purpose, the ID of the WF instance may be a good choice. Alternatively, its starting time (or a special random value generated for this purpose) may be used. If, for example, the value representing the milliseconds of the starting time is used as *InstDat*, it can be assumed that the values of *InstDat* are equally distributed within the interval $[0, 1000)$. For this case, the function $hash(InstDat) := InstDat/1000$ will be sufficient to achieve an equal distribution of z in the interval $[0, 1)$.

3.4.2. *Replication of the Distribution Functions*

When starting or migrating a WF instance *Inst*, one must be able to decide, which WF server s of the domain D should control this WF instance. For this, the function $g^D(Inst)$ has to be published to all servers of the WfMS. Since g^D does only calculate logical server IDs, its output has to be mapped to a physical address (e.g., an IP address) by another function $h(s)$, which must also be made known to all WF servers. As both functions are changed seldomly (cf. Section 4), the communication costs arising in conjunction with their replication can be neglected.

4. Dynamic Modification of the Load Distribution

The hash approach, described so far, already fulfills the Requirements 1-4. The changeability of the load distribution (Requirement 5) has not been considered yet. Such a changeability is indispensable to overcome overload situations or to modify the system configuration.

A change may be triggered by an administrator or automatically by a "watch dog" at a dedicated administration server. Other servers, however, may be involved in the change as well. Since a WF instance always has to be controlled by the same server within one domain, it is really hard to realize such a change of the load distribution. Unfortunately, it requires a modification of the assignment between WF instances and servers. This, in turn, requires additional communication in order to publish the changed assignment to all WF servers. Since corresponding modifications are rather seldom and the communicated data volumes are usually rather small, this additional effort is small when compared to the "regular" execution of workflows. In the following, we show how WF servers may be replaced, added, or removed in our approach. In addition, we discuss how a load distribution may be changed on the fly. All these operations are necessary in order to fulfill Requirement 5. The challenge is to realize them without violating the Requirements 1-4.

4.1. *Replacing Physical WF Servers*

We shortly present an algorithm (cf. Algorithm 2) for replacing physical WF servers in our approach. Assume, the WF server with the logical ID s_{change} has to be replaced by another one. Then, first of all, the new server is started and the old server s_{change} is locked for further migrations. In order to realize this lock, all WF servers have to be informed that migrations to s_{change} are no longer possible. In such a case, respective migrations are delayed until the lock is released. In addition, s_{change} prevents the (local) creation of new WF instances. It is important to mention that this procedure does not imply any restrictions for migrations to other WF servers.

After having locked s_{change} for further migrations, the function $h(s)$, which maps logical server identifiers to physical addresses, is changed. Afterwards, all WF instances, which have been controlled by s_{change} thus far, are transferred to the new WF server. Furthermore, the newly created function $h_{(n+1)}(s)$ is replicated to all servers of the WfMS, substituting the current function $h_{(n)}(s)$. Consequently, no migrations will arrive at s_{change} any longer. When finishing these steps, the locks are released in order to enable migrations to the new server. Finally, s_{change} is stopped.

Algorithm 2 (Replacement of a WF Server).

```
input
    $s_{change}$:   logical ID of the WF server to be replaced
    $adr$:   physical address of the server, which has to obtain the logical ID $s_{change}$
    $h_{(n)}(s)$:   the current function that maps the logical server IDs to physical addresses
begin
    start the new WF server $adr$;
    lock migrations (at all WF servers) to the server $s_{change}$;
    // assign the new server ID to $s_{change}$, leave all other mappings unchanged
    $h_{(n+1)}(s_{change}) := adr$; $\forall s \neq s_{change}$: $h_{(n+1)}(s) := h_{(n)}(s)$;
    transfer all WF instances from the server $h_{(n)}(s_{change})$ to the server $h_{(n+1)}(s_{change})$;
    replicate $h_{(n+1)}(s)$ to all WF servers;
    release the locks for migrations to the WF server $s_{change}$;
    stop the WF server $h_{(n)}(s_{change})$;
end.
```

Theoretically, the lock-request, the replication of $h_{(n+1)}(s)$, and the lock-release should be performed within one distributed transaction. As long as this transaction has not been completed at all WF servers, migrations to the server s_{change} would not be possible. Therefore, the breakdown of any server involved in the execution of Algorithm 2 would block the server s_{change}. In addition, such a procedure would require the execution of an expensive 2-phase-commit protocol.

In order to avoid these disadvantages, Algorithm 2 and the subsequently described algorithms use the following technique: Again, all WF servers prohibit outgoing migrations to s_{change}. After the execution of the corresponding lock-request, no migration will arrive at the server s_{change}. The replication of the function $h_{(n+1)}(s)$ and the release of the lock are merged into a single transaction: As soon as a server has saved the new data (locally), it releases its local lock for migrations to the server s_{change}. Hence, it may perform corresponding migrations based on the new function $h_{(n+1)}(s)$. Using this technique, the locks are not released simultaneously at the different WF servers. The possibility to migrate, however, is still locked at a server or the server uses the new function $h_{(n+1)}(s)$. Therefore, no inconsistencies may occur due to the use of the different functions $h_{(n)}(s)$ and $h_{(n+1)}(s)$. The crucial advantage of this technique is that each WF server may continue its work without any restrictions, after having successfully stored the data (locally). This will even be valid, if not all servers have performed the modification of the mapping function so far, as in the case of a local system breakdown, for example.

4.2. *Changing the Load Distribution*

Assume that the distribution of the load between the WF servers of a domain D has to be modified, without introducing new servers to this domain. Then, the current distribution function $g^D_{(n)}(\mathit{Inst})$ has to be replaced by a new one $g^D_{(n+1)}(\mathit{Inst})$. Such a replacement, however, is non-trivial in the context of WF instances with parallel branchings. If it is applied in an unsynchronized manner to the WF servers, some migrations of a WF instance may be based on the old function while others may use the new one. Thus, a WF instance might be controlled by different WF servers within the same domain. Obviously, this should be prevented, in order to avoid unnecessary communication between the WF servers of a domain (Requirement 2). This can be achieved by applying one of the following two approaches:

1. First we consider a time-based approach (cf. Algorithm 3). It assigns a time-stamp $T^D_{(n+1)}$ to the new distribution function $g^D_{(n+1)}(\mathit{Inst})$ of the domain D, which is replicated together with $g^D_{(n+1)}(\mathit{Inst})$. The newly introduced function $g^D_{(n+1)}(\mathit{Inst})$ is only used for WF instances *Inst* with a creation time greater than $T^D_{(n+1)}$, while other instances further use $g^D_{(n)}(\mathit{Inst})$. It is important to mention that migrations to the domain D are prevented until Algorithm 3 is finished. When migrating a WF instance with $\mathit{CreationTime} > T^D_{(n+1)}$ to the domain D, therefore, the new function $g^D_{(n+1)}(\mathit{Inst})$ and the corresponding time-stamp will have been already known to this domain. Following this approach, it is always possible to perform corresponding migrations on the basis of the new function $g^D_{(n+1)}(\mathit{Inst})$.

2. An alternative method is to calculate the minimal set of WF instances, for which a change of the distribution function may cause problems. For these instances, the distribution of the load is further based on the old distribution function. Algorithm 4 calculates the set $A^D_{(n+1)}$ which contains the IDs of these WF instances. Each of them is characterized by the following properties:

   a) The instance is controlled by a WF server of the domain D while the distribution function is modified.

Algorithm 3 (First Method for the Modification of $g^D(Inst)$).

```
input
    $D$:   domain for which the load distribution has to be changed
    $g^D_{(n+1)}(Inst)$:   new load distribution function of the domain $D$
begin
    lock migrations (at all WF servers) to the domain $D$;
    $T^D_{(n+1)} = time()$;    // set the time-stamp to the current time
    replicate $(g^D_{(n+1)}(Inst), T^D_{(n+1)})$;
    release the locks for migrations to the domain $D$;
end.
```

b) For this WF instance, the modification would result in a change of the corresponding WF server within the domain D.

The new load distribution function $g^D_{(n+1)}(Inst)$ is not valid for WF instances $Inst$ with $ID(Inst) \in A^D_{(n+1)}$. Instead, they further use the old function $g^D_{(n)}(Inst)$ (or $g^D_{(n-1)}(Inst)$ if $ID(Inst) \in A^D_{(n)}$ additionally holds, etc.). Consequently, we can ensure that all parallel branches of such a WF instance are always controlled by the same server with respect to a given domain D. For all other WF instances (with $ID(Inst) \notin A^D_{(n+1)}$), solely the "new" function $g^D_{(n+1)}$ is applied; i.e., also for these WF instances no problems will occur when joining parallel branches.

Algorithm 4 (Second Method for the Modification of $g^D(Inst)$).

```
input
    $D$:   domain for which the load distribution has to be changed
    $g^D_{(n+1)}(Inst)$:   new load distribution function of the domain $D$
    $S_D$:   WF server set of the domain $D$
begin
    lock migrations (at all WF servers) to the domain $D$;
    $activeWFs = \bigcup_{s \in S_D} \{Inst \mid \text{the WF instance } Inst \text{ is active at the WF server } s\}$;
    // WF instances, which are active in the domain $D$, for those the WF server would
    // change:
    $A^D_{(n+1)} = \{ID(Inst) \mid Inst \in activeWFs \wedge Server^D(Inst) \neq g^D_{(n+1)}(Inst)\}$;
    replicate $(g^D_{(n+1)}(Inst), A^D_{(n+1)})$;
    release the locks for migrations to the domain $D$;
end.
```

The time-stamp-based method only requires a very small effort. The new distribution function $g^D_{(n+1)}(Inst)$, however, may not be applied to all WF instances as long as there are ones, which had been created before this function was introduced. Since WF instances may be long-running (up to several weeks or months), it may take a long time to reach the intended load distribution. Although, the second method requires a larger effort than the first one (due to the calculation and replication of $A^D_{(n+1)}$), it has the advantage that the new load distribution function can be used for already running WF instances as well. Which one of the two methods is more suitable, depends on the respective goals. If the load distribution has to be adjusted only for a longer term, the first method is sufficient. If the effects of the modification must be observed and evaluated immediately, however, the long delay coming with this time-stamp-based approach cannot be accepted. For this case, the second method has to be applied. The resulting costs are acceptable as for most scenarios, a modification of the load distribution is required only seldom.

4.3. *Introducing Additional Servers*

If a new WF server has to be installed for a given domain D, first a logical server ID s_{new} must be defined. Afterwards, the new WF server is started and its logical server ID s_{new} is mapped to its physical address adr_{new} (cf. Algorithm 5). This is realized by changing the function $h(s)$ and by replicating it. Initially, no interval of the distribution function is assigned to s_{new}. It, therefore, will not be used immediately. Thus, the procedure described by Algorithm 5 does not require any locks. In order to enable the new server to control WF instances, first of all, the function $g^D(Inst)$ has to be modified (i.e., an interval of the load distribution function has to be assigned to the new WF server) as described in the previous section.

Algorithm 5 (Introduction of a New WF Server).

```
input
    $s_{new}$:   logical ID of the new WF server
    $adr_{new}$:   physical address of the new WF server
    $h_{(n)}(s)$:   current function, that maps logical server IDs to physical server addresses
    $g^D_{(n'+1)}(Inst)$:   new distribution function of the domain $D$ which respects the server
                    $s_{new}$ as well
begin
    start the WF server $adr_{new}$;
    $h_{(n+1)}(s_{new}) := adr_{new}$; $\forall s \neq s_{new}$: $h_{(n+1)}(s) := h_{(n)}(s)$;
    replicate $h_{(n+1)}(s)$;
    change $g^D_{(n')}(Inst)$ to $g^D_{(n'+1)}(Inst)$ using Algorithm 3 or 4;
end.
```

4.4. *Removing Servers from a Domain*

If a WF server s_{old} has to be removed from the domain D, first of all, it must be locked for further migrations (cf. Algorithm 6). Following this approach, all WF instances controlled by s_{old} are subsequently transferred to the server s_{new}, which (temporarily) takes over its tasks. In order to prevent incoming migrations at the "old" server, its logical server ID s_{old} is "redirected" to the physical address of the "new" one. This is achieved by changing the function $h_{(n)}(s)$. Afterwards, the resulting function $h_{(n+1)}(s)$ is replicated and all locks are released.

Following this approach, the old server no longer controls any WF instances. In addition, no WF instance may migrate to this server in the future. The server, therefore, can be stopped. In order to remove its logical server ID, after the completion of Algorithm 6, the load distribution function should be modified as well. For this, the methods presented in Section 4.2 can be applied, such that no interval is assigned to s_{old} anymore. Furthermore, the load that was managed by the server s_{old}, thus far, may be distributed to the other WF servers by adjusting all intervals adequately.

The load distribution function may be realized efficiently and with good statistical properties. As we have shown, it is even possible to change this function in a

```
Algorithm 6 (Remove a WF Server).
input
    $s_{old}$: logical ID of the WF server to be removed
    $s_{new}$: logical ID of the WF server, that takes over the tasks of $s_{old}$
    $h_{(n)}(s)$: current function, that maps logical server IDs to physical server addresses
begin
    lock migrations (at all WF servers) to the server $s_{old}$;
    $h_{(n+1)}(s_{old}) := h_{(n)}(s_{new})$; $\forall s \neq s_{old}$: $h_{(n+1)}(s) := h_{(n)}(s)$;
    transfer all WF instances from the server $h_{(n)}(s_{old})$ to the server $h_{(n)}(s_{new})$;
    replicate $h_{(n+1)}(s)$;
    release the locks for migrations to the server $s_{old}$;
    stop the WF server $h_{(n)}(s_{old})$;
end.
```

running WfMS arbitrarily. Our approach, therefore, does not only fulfill Requirements 1-4 – as already shown in Section 3 – but also Requirement 5 (cf. Section 1.2).

5. Simulation

To verify the effectiveness of the presented methods, we performed numerous simulations. For this purpose, we had realized a program that simulates the arrival of WF instances at the WF servers of the inspected domain and, in addition, their retention at these WF servers.

The simulation considers multiple WF types, whereas the corresponding WF instances cause different loads. In addition, these WF instances have an unequal starting frequency. It is assumed that their arrival at an inspected partition is distributed equally over time. Since no influence on the results of the simulations is expected, to simplify matterns, we further assume that the retention periods of these WF instances are also distributed equally over a certain period of time. For the different WF types, however, this time period is non-uniform.

For the simulation time of one year, the load behavior of the different WF servers that belong to the inspected domain is recorded. These data are used as basis for the analyses presented in the following. In order to allow an evaluation of the statistical quality of the results, the simulation was performed 10 times. We observed, thereby, that the result of the simulation is very exact; for all presented values the 99% confidence interval is smaller than $\pm 1\%$ of the value itself. The most interesting parameter of the performed simulations is the *number of WF instances* which had been concurrently active in the inspected domain. As expected, the deviations from the intended load will be smaller if this number becomes larger (cf. Fig. 5). In addition, even the maximal and minimal deviations from the intended load, which occur during the simulation, become very small, when simulating a large number of WF instances.

As an example, we present the results of one simulation in order to show that the deviations from the intended load are extremely small if a reasonable number of concurrent WF instances is simulated. We consider a simulation for a domain with 3 WF servers. For this domain, we assume that the available capacities of the corresponding server machines make it necessary that the load is distributed with the ratio 20% : 30% : 50% to them. The simulation treats WF instances of 4 different WF types, whereas each WF type possesses different attributes (e.g., with respect to starting frequencies, generated loads, or retention times of corresponding WF instances). Four simulation variants have been considered, each of them with a different number of concurrently active WF instances (on the average) within the domain.

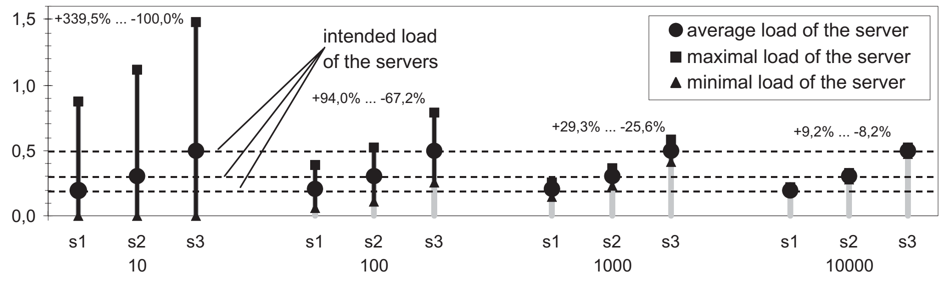

Figure 5: Result of the simulation for 10, 100, 1000, and 10000 concurrently active WF instances within a domain. The numbers above the bars indicate the relative deviation of the maximal and minimal load from the average value.

The simulation results are illustrated by Fig. 5. The y-axis denotes the part of the total load a server has to handle. This load portion corresponds to the part of the communication costs, processing costs, and other costs this server has to cope with. Note that WF servers of the same domain are homogenous. For all simulated scenarios, the average load of each WF server corresponds to the intended load very well. Furthermore, the minimal and the maximal load occuring at a WF server (within the simulated period of one year) heavily depend on the number of WF instances concurrently active in the corresponding domain. The maximal load of a WF server may exceed the value 1.0 if this server has to handle a load (at one point in time) that is larger than the average total load of the domain. For very small systems (e.g., a domain with only 10 concurrently active WF instances is unrealistically small), great variations may occur with respect to these loads. As opposed to this, for large systems with 10,000 or more concurrently active WF instances, even the maximum variations are relatively small, namely less than $\pm 10\%$ of the intended load.

The presented simulation only analyses the load situation of the servers for different numbers of WF instances concurrently active in the domain (on average). This is permissible since almost all other simulation parameters (e.g., the number of activities of the WF instances of a particular WF type or the loads generated by these activities) have no influence on the overall simulation result. They only concern one single WF type, and, therefore, merely influence the load that is created when executing WF instances of this type. Nevertheless, for this load we assume anyway, that it varies from WF type to WF type. The same holds for the created number of WF instances (of the different WF types). In the following we argue that the load actually created by the instances of a WF type, in fact, has no influence on the result depicted in Fig. 5.

Fig. 6 shows that for each WF type, the part of the WF instances controlled by a given WF server, corresponds to its intended value very precisely (s_1: 20 %, s_2: 30%, and s_3: 50% in our example). For several reasons, it is crucial that this is also valid with respect to the instances of each individual WF type, and not only for the total number of WF instances. First of all, WF instances of different type may generate an unequal load. In the given simulation scenario, for example, we have assumed that each WF instance of Type 1 creates one load unit, each instance of Type 2 three load units, each instance of Type 3 two load units, and each instance of Type 4 four load units. Secondly, WF instances of different types may be started with different frequencies (in our example: Type 1: 30%, Type 2: 25%, Type 3: 20%, and Type 4: 25% of the total number of WF instances started). In any case, the resulting load is distributed (almost) identically to the different WF servers –

for each individual WF type – as can be seen from Fig. 6. It is allowed, therefore, to inspect the WF types jointly. That means, the fact that the different WF types cause an unequal load (and how big it is) has no influence on the total loads of the different WF servers depicted in Fig. 5. The same holds for the number of activities of the individual WF types, the costs of single activities, the number of their potential actors, and other parameters belonging to a WF type, because they only affect the load created by the WF instances.

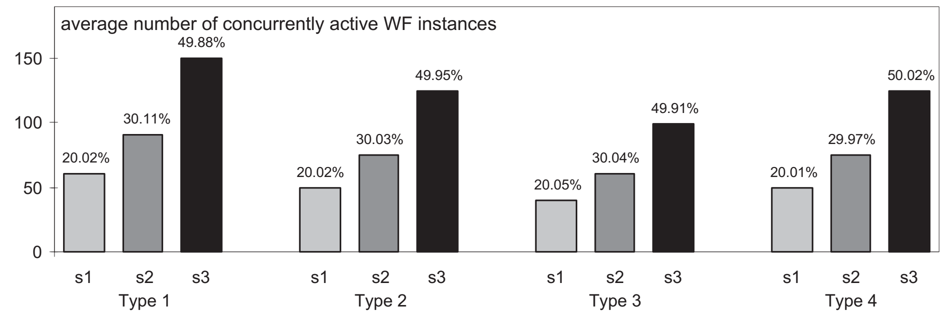

Figure 6: Load of the WF servers for 1000 concurrently active WF instances (depicted separately for the different WF types).

In summary, our simulations have shown that the presented approach will allow the load of a WF server to be kept very close to the intended value, if the WfMS is sufficiently burdened. The presented methods, therefore, work very well.

6. Related Work

In this section we discuss work that is related to the presented approach. In detail, we have a closer look at scheduling methods for distributed operating system processes and at approaches for distributed WF management.

6.1. *Process Scheduling*

As already sketched at the end of Section 3.2, process scheduling in distributed operating systems shows similarities to the assignment of activity instances to WF servers.

6.1.1. *Scheduling of Processes in Distributed Operating Systems*

Distributed operating systems use load balancing algorithms to determine the processor to which a particular operating system process shall be assigned. First of all, it is important to mention that operating system processes do not correspond to the WF instances of a WfMS. Usually, a WF instance consists of several activity instances, which have to be assigned to the WF servers. In the same way, an application consists of operation system processes, which are assigned to the processors. Consequently, the operating system processes correspond to the activity instances. The processor, however, is normally determined when starting an operating system process (non-pre-emptive scheduling [24]). As a consequence, no running processes have to be transferred. Note that, similar to a migration, this would be an expensive operation.

The goal of *dynamic scheduling methods* is to distribute the overall load to the available processors such that they are equally burdened. These methods build the left sub-tree of the classification of global load balancing methods as depicted in Fig. 7 (see also ^{24,25}). (*Local scheduling* denotes the distribution of the time slots of one single processor to the local processes.) Dynamic load balancing algorithms may be physically distributed or may be centralized. For *centralized load balancing algorithms*, a central scheduler decides which processor has to execute which process. One disadvantage of this approach is that for each processor, information about its current load has to be transmitted to the central scheduler. In addition, this central component represents a bottleneck, therefore, affecting system availability. Concerning *distributed scheduling methods*, the decision is made jointly by multiple processors. For this purpose, the locally available information about the load level of the processors is used. Distributed scheduling algorithms may be *cooperative*; i.e., the different processors pursue a common goal. Contrary to this, *non-cooperative* scheduling methods optimize the performance of the local processor. This may have the consequence, however, that different processors make contradictory decisions. Cooperative scheduling algorithms may achieve an optimal or sub-optimal result. A sub-optimal solution arises if approximative methods or heuristics are used.

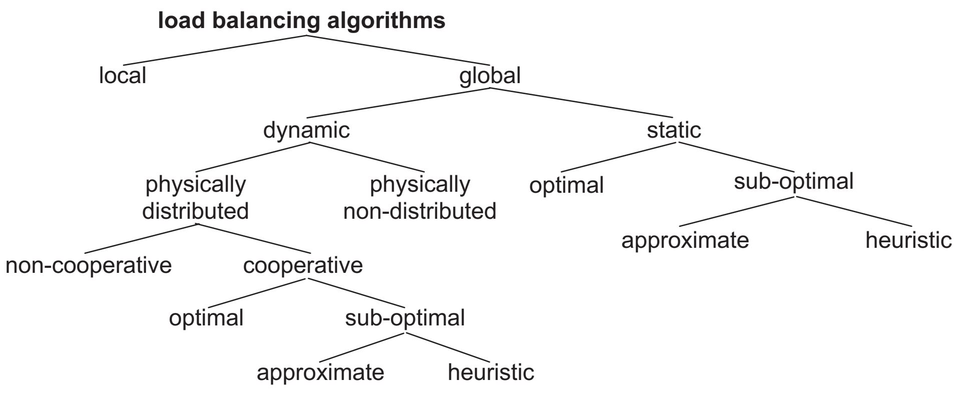

Figure 7: Hierarchical classification of process scheduling methods.

Static scheduling methods (cf. Fig. 7) assume that the resource requirements of the operating system processes are known before they are started ^{24,25}. This allows to select a processor that matches well to the properties of the process. Two variants of such methods are distinguished: The first one calculates the *optimal* distribution of the processes to the processors. The second one achieves only a *sub-optimal* solution, which is determined by the use of *approximation techniques* or *heuristics*. For the calculation of the optimal solution and for the approximation of a sub-optimal solution, the same techniques are used, namely the (partially) scanning of the possible solutions.

6.1.2. *Comparison with Load-balancing in WfMS*

Let us now discuss, how the presented load-balancing approach for WfMS is related to these process scheduling methods. In addition to the general remarks given in Section 3.2, the following statements can be made: In WfMS much useful information is available about the WF types (e.g., the potential actors of the activities and the subnets these users belong to). The situation, therefore, corresponds to that of static scheduling methods. ADEPT uses this information to determine well-suited domains for the WF partitions (cf. Section 1.1). For the selection of a concrete WF server within a given domain, in principle, a dynamic load balancing method is applied. When such methods are used, normally, the problems described in Section 3.3

occur in conjunction with parallel branches, and also in conjunction with operations, that involve all servers currently controlling a WF instance. (An example for such an operation is the dynamic modification of a WF instance in a distributed WfMS [16]. For several reasons, in such a case, all servers currently controlling the modified WF instance have to learn the WF instance graph structure resulting from the change.) To solve these problems, a load balancing method (random selection of the servers) was extended that way that for one WF instance always the same WF server is chosen. The result is the presented hashing approach.

6.2. *Distributed Workflow Management Systems*

Although issues related to distributed WF management are considered as very important in the WF literature [17,26,27,20,21,8,22,11,19], to our best knowledge, so far, there has been no work on distributed WF management that deals with the application of randomized methods to select the WF server for a given WF partition. For the first time, this paper examines the problems, which result from the use of several WF servers within one domain. Additionally, it shows how an appropriate server can be selected for activity executions in such a scenario. In more detail, this section offers a short overview of distribution models for WfMS as suggested in the WF literature (see Fig. 8). We discuss, how the different distribution models are related to the ADEPT approach and how the concepts presented in this paper may be used in combination with these models.

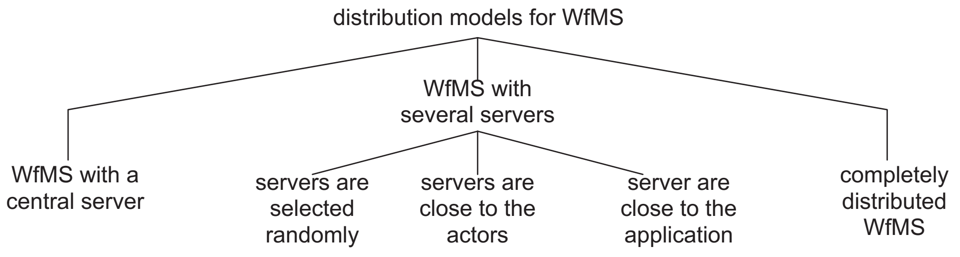
distribution models for WfMS
WfMS with several servers
WfMS with a central server
servers are selected randomly
servers are close to the actors
server are close to the application
completely distributed WfMS

Figure 8: Classification of distribution models for WfMS.

Some WF research projects, which have not been primarily concerned with scalability issues (e.g., Panta Rhei [28] and WASA [29]), and most commercial WfMS (e.g., Staffware [30]) use a central WF server. Since such a central server represents a bottleneck, these WfMS are not scalable.

Completely distributed approaches (e.g., Exotica/FMQM [26] and INCAS [27]) have no central WF server. Instead they realize the WF server functionality by the use of the actor machines; i.e., a particular WF instance is always controlled by the computer of that user who currently works on an activity of this instance. One disadvantage of such approaches is that the synchronization of the potential actors who may work on an activity is very expensive since an additional distributed protocol is required. Furthermore, for (almost) each activity execution, a migration of the WF instance to the computer of the corresponding actor becomes necessary, which results in high communication costs. Of course, using such a distribution model, there is no central WF server (in the conventional sense) which could represent a potential bottleneck. Consequently, no WF server may become overloaded, such that there is no need for using our approach.

In the Exotica project, an approach was developed, which is based on the random choice of a WF server (cluster) at the time a WF instance is started [17]. The WF instance remains in the selected cluster during its whole lifetime; i.e., there are no migrations (as in ADEPT). As already discussed in Section 2, such a distribution at the WF instance level may be unfavorable if the potential actors of the activities are geographically spread. For this case, it is generally not possible to achieve locality between actor and server for all WF activities. Strictly speaking, this distribution model corresponds to a central approach with a replicated WF server, whereas the actual server for the control of a particular WF instance is randomly chosen (cf. Section 3.3). Since a WF instance never changes the cluster, no problems occur when joining parallel branches.

Many approaches use several WF servers, allowing that WF instances may migrate from one WF server to another during run-time. These approaches use different strategies for partitioning WF graphs and for selecting the concrete WF server of a partition. As ADEPT, MENTOR [8] and WIDE [20] follow an *actor-centric* approach. They choose the WF server "near" to the potential actors of the current activity. Unlike ADEPT, these two approaches have the limitation that all potential actors of an activity must belong to the same domain as the server of the activity. This may be contradictory to organizational facts. In addition, it may result in unprofitable migrations (of the whole WF instance) if single activities are performed by actors of another domain. METEOR$_2$ [21], CodAlf [22], and BPAFrame [22] choose the server near to the application that belongs to the current activity. This *application-centric* view may lead to an unfavorable communication behavior for the users, because the server is not necessarily located in the subnet of the current actors. For all these distribution approaches, a WF server may become overloaded. The presented hash approach would allow to replicate such a WF server and to distribute the load to the resulting servers.

MOBILE [18] assigns the instances of distinct WF types to different WF servers. Migrations are not supported. A sub-process, however, may be controlled by a different WF server [19]. This server is chosen at run-time, based on diverse criteria (e.g., rights and weights). Some commercial products (e.g., IBM MQSeries Workflow [3,31]) enable remote sub-process execution as well. Principally, this corresponds to an approach with a central WF server and an extension for distributed WF execution. Its disadvantage is that each modification of the server assignment also requires an expensive change of the WF model. Furthermore, following such an approach, a WF server may become overloaded as well. In this context, our approach could be used to solve this problem. As only whole sub-processes may be controlled by other servers, parallel branches are always joined by that server that controls the corresponding split node. Therefore, no problem (cf. Section 3.3) may occur when joining parallel branches. Thus, it is also possible to use the suggested variant with a random server selection.

There is another promising approach [32] that is orthogonal to the distribution model of the WfMS. It uses continuous-time Markov chains to calculate the most suitable configuration of a distributed WfMS; i.e., the optimal number of instances of each system component. In this work, the performance and the availability of the WfMS have been considered, whereas issues related to the resulting communication costs have not been discussed. In our scenario, a similar model could be used to calculate and adapt the intended load of the individual WF servers.

To summarize, the application of the presented methods is not restricted to ADEPT. They may be advantageous for many other approaches as well. For approaches that do not use migrations, a randomly chosen server is sufficient. Otherwise, the hash variant is best suited (as for ADEPT).

7. Summary and Outlook

The numerous tasks of a WF server may result in its overload. This can be prevented if the WF server is replicated and the load is distributed to the resulting servers. In this paper, we have presented an approach that hashes WF instances to the WF servers of a domain. By means of a simulation we have shown that this approach enables an arbitrary and definable distribution of the load to the WF servers. Basically, the approach does not generate any additional communication. Since only a simple hash function has to be evaluated, it requires a very small computational effort. Furthermore, it allows to change the distribution of the load between the WF servers while the system is running. Thus, the problem of overloaded WF servers is solved.

In this paper, we have also discussed the applicability of load-dependent approaches for WF server selection. In general, these approaches have many disadvantages when compared to hashing. But they may be relevant for some special applications. This will be the case if the available capacity of the WF servers fluctuates, if the breakdown of WF servers has to be compensated by these methods, or if a WF server is only able to control very few WF instances. In the latter case, even a small unequal distribution of the load, as it may be caused by the hash approach, would result in difficulties.

There are some topics, which are related to distribution aspects. Since we have focused on performance issues, reliability and availability aspects have not been discussed in this paper. They are independent from our work, because they concern the use of backup servers [4]. In addition, ADEPT assumes a homogenous server environment. To address interoperability questions, it should be examined if the presented approach is also applicable in a heterogeneous environment, where remote invocations of sub-workflows have to be used instead of migrations.

Acknowledgements: We would like to thank the anonymous reviewers and our colleagues Clemens Hensinger and Thomas Fries for their valuable suggestions.

1. W.M.P. van der Aalst and K. van Hee. *Workflow Management.* MIT Press, 2002.
2. L. Fischer. *Workflow Handbook 2001.* Future Strategies Inc., 2000.
3. F. Leymann and D. Roller. *Production Workflow - Concepts and Techniques.* Prentice Hall, 2000.
4. M. Kamath, G. Alonso, R. Günthör, and C. Mohan. Providing High Availability in Very Large Workflow Management Systems. In *Proc. 5th Int. Conf. on Extending Database Technology*, Avignon, 1996.
5. T. Bauer and P. Dadam. Distribution Models for Workflow Management Systems – Classification and Simulation. *Informatik Forschung und Entwicklung*, 14(4), 1999. (in German).
6. T. Bauer and P. Dadam. Efficient Distributed Workflow Management Based on Variable Server Assignments. In *Proc. 12th Conf. on Advanced Inf. Syst. Engineering*, Stockholm, 2000.
7. T. Bauer and P. Dadam. A Distributed Execution Environment for Large-Scale Workflow Management Systems with Subnets and Server Migration. In *Proc. 2nd IFCIS Conf. on Cooperative Inf. Syst.*, Kiawah Island, SC, 1997.
8. P. Muth, D. Wodtke, J. Weißenfels, A. Kotz-Dittrich, and G. Weikum. From Centralized Workflow Specification to Distributed Workflow Execution. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, 10(2), 1998.
9. T. Bauer. *Efficient Realization of Enterprise-wide Workflow Management Systems.* PhD thesis, Universität Ulm, Fakultät für Informatik, 2001. (Tenea-Verlag, in German).
10. P. Dadam and M. Reichert. *Proc. Workshop Enterprise-wide and Cross-Enterprise Workflow Management, Concepts, Systems, Applications.* 29. Jahrestagung der GI, 1999.
11. A. Sheth and K.J. Kochut. Workflow Applications to Research Agenda: Scalable and Dynamic Work Coordination and Collaboration Systems. In *Proc. NATO Advanced*

Study Institute on Workflow Management Systems and Interoperability, Istanbul, 1997.

12. P. Dadam, M. Reichert, and K. Kuhn. Clinical Workflows - The Killer Application for Process-oriented Information Systems? In *Proc. 4th Int. Conf. on Business Inf. Syst.*, Posen, 2000.
13. M. Reichert and P. Dadam. ADEPT$_{flex}$ – Supporting Dynamic Changes of Workflows Without Losing Control. *Journal of Intelligent Information Systems, Special Issue on Workflow Management Systems*, 10(2), 1998.
14. H. Enderlin. Realization of a Distributed Workflow Execution Component Based on IBM FlowMark. Master's thesis, Universität Ulm, Fakultät für Informatik, 1998. (in German).
15. T. Bauer and P. Dadam. Efficient Distributed Control of Enterprise-Wide and Cross-Enterprise Workflows. In *Proc. Workshop Enterprise-wide and Cross-enterprise Workflow Management: Concepts, Systems, Applications, 29. Jahrestagung der GI*, Paderborn, 1999.
16. T. Bauer, M. Reichert, and P. Dadam. Supporting Ad-hoc Changes in Distributed Workflow Management Systems. *Datenbank-Spektrum*, 1(1):68–77, September 2001. (in German).
17. G. Alonso, M. Kamath, D. Agrawal, A. El Abbadi, R. Günthör, and C. Mohan. Failure Handling in Large Scale Workflow Management Systems. Technical Report RJ9913, IBM Almaden Research Center, 1994.
18. P. Heinl and H. Schuster. Towards a Highly Scaleable Architecture for Workflow Management Systems. In *Proc. 7th Int. Workshop on Database and Expert Systems Applications*, Zurich, 1996.
19. H. Schuster, J. Neeb, and R. Schamburger. A Configuration Management Approach for Large Workflow Management Systems. In *Proc. Int. Joint Conf. on Work Activities Coordination and Collaboration*, San Francisco, 1999.
20. S. Ceri, P. Grefen, and G. Sánchez. WIDE – A Distributed Architecture for Workflow Management. In *Proc. 7th Int. Workshop on Research Issues in Data Engineering*, Birmingham, 1997.
21. S. Das, K. Kochut, J. Miller, A. Sheth, and D. Worah. ORBWork: A Reliable Distributed CORBA-based Workflow Enactment System for METEOR$_2$. Technical Report #UGA-CS-TR-97-001, Department of Computer Science, University of Georgia, 1997.
22. A. Schill and C. Mittasch. Workflow Management Systems on Top of OSF DCE and OMG CORBA. *Distributed Systems Engineering*, 3(4), 1996.
23. A.S. Tanenbaum. *Computer Networks*. Prentice-Hall, 1996.
24. T.L. Casavant and J.G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, SE-14(2):141–154, 1988.
25. A. Goscinski. *Distributed Operating Systems: The Logical Design*. Addison-Wesley, 1991.
26. G. Alonso, C. Mohan, R. Günthör, D. Agrawal, A. El Abbadi, and M. Kamath. Exotica/FMQM: A Persistent Message-Based Architecture for Distributed Workflow Management. In *Proc. IFIP Working Conf. on Inf. Syst. for Decentralized Organisations*, Trondheim, 1995.
27. D. Barbará, S. Mehrotra, and M. Rusinkiewicz. INCAs: Managing Dynamic Workflows in Distributed Environments. *Journal of Database Management*, 7(1), 1996.
28. J. Eder, H. Groiss, and W. Liebhart. Workflow Management and Databases. In *2ème Forum Int. d'Informatique Appliquée*, Tunis, 1996.

29. M. Weske. Flexible Modeling and Execution of Workflow Activities. In *Proc. 31st Hawaii Int. Conf. on System Sciences*, pages 713–722, Hawaii, 1998.
30. Staffware. *Server Administrators Guide*, 1999.
31. IBM. *MQSeries Workflow Administrators Guide*, 1999.
32. M. Gillmann, J. Weissenfels, G. Weikum, and A. Kraiss. Performance and Availability Assessment for the Configuration of Distributed Workflow Management Systems. In *Proc. 7th Int. Conf. on Extending Database Technology*, Konstanz, 2000.