Peer–to–Peer Process Execution with OSIRIS

Christoph Schuler[1], Roger Weber[1], Heiko Schuldt[2], and Hans-J. Schek[1]

[1] Swiss Federal Institute of Technology (ETH),
CH–8092 Zurich, Email: {schuler,weber,schek}@inf.ethz.ch
[2] University for Health Informatics and Technology Tyrol (UMIT),
A-6020 Innsbruck, Email: heiko.schuldt@umit.at

Abstract. Standards like SOAP, WSDL, and UDDI facilitate the proliferation of services. Based on these technologies, processes are a means to combine services to applications and to provide new value-added services. For large information systems, a centralized process engine is no longer appropriate due to limited scalability. Instead, in this paper, we propose a distributed and decentralized process engine that routes process instances directly from one node to the next ones. Such a Peer-to-Peer Process Execution (P^3E) promises good scalability characteristics since it is able to dynamically balance the load of processes and services among all available service providers. Therefore, navigation costs only accumulate on nodes that are directly involved in the execution. However, this requires sophisticated strategies for the replication of meta information for P^3E. Especially, replication mechanisms should avoid frequent accesses to global information repositories. In our system called OSIRIS (*Open Service Infrastructure for Reliable and Integrated Process Support*), we deploy a clever publish/subscribe based replication scheme together with freshness predicates to significantly reduce replication costs. This way, OSIRIS can support process-based applications in a dynamically evolving system without limiting scalability and correctness. First experiments have shown very promising results with respect to scalability.

1 Introduction

Modern technologies like XML, SOAP, and WSDL provide a simple yet powerful means to publish information services and to access services. The platform independent definitions of these technologies further simplify the composition of services to offer new value added services [5]. One way to achieve this goal is to define transactional processes [20] over web services. Such processes compose web service calls in an application-specific invocation order –by defining control and data flow– together with transactional guarantees. Each activity of a process corresponds to the invocation of a (web) service. Programming using processes is referred to as "programming in the large" or "mega programming" [24]. Essentially, processes are again (higher-level) web services, i.e., they are accessible via SOAP (Simple Object Access Protocol) [21] and described by a WSDL (Web Service Description Language) [26] document such that other users can easily integrate them into even larger processes.

Processes impose several vital requirements to the underlying infrastructure. In terms of failure handling, the traditional "all-or-nothing" semantics of atomicity is too restrictive and should be generalized. Process specification must allow for several contingency strategies to handle failures at the application level. Essentially, it is the task of an infrastructure to guarantee correctness even in case of network or software failures. Although not all applications require concurrency control, an infrastructure should be able to coordinate the access to shared resources in case it is needed.

Usually, several semantically equivalent web services are available at different places. An infrastructure for processes must be able to equally distribute the load over all web service providers. Similarly, we want to optimize process executions by taking costs and expected execution times. To this end, the infrastructure should bind services dynamically at execution time rather than at process definition time. This decoupling provides a high degree of flexibility since new providers and services are seamlessly integrated (following the ideas of autonomic computing [25]).

Finally, the system has to provide a high degree of scalability in terms of the number of service providers and the number of concurrent processes.

1.1 Survey of Existing Infrastructures

Recently, various *frameworks* supporting the invocation of web services have been proposed. Microsoft's .NET [14], for example, allows to integrate web service calls into applications. However, since each web service call is considered independently, .NET does not provide execution guarantees for complete applications. Moreover, application developers must link services at build-time and there is no support for dynamic service selection at run-time. Therefore, the task of balancing the load among all available service providers is shifted from the infrastructure to the applications.

Workflow management systems are the traditional infrastructures for process support. Systems like IBM's MQSeries Workflow [13] support the integration of web service calls into workflow processes. Usually, these systems offer well-engineered execution guarantees, especially in terms of failure handling. However, they lack a flexible binding of service calls at run-time and, thus, are not able to optimally distribute the overall load among all service providers at run-time. Since these systems follow a centralized architecture consisting of dedicated workflow engine(s), their scalability is limited.

In contrast, *grid* infrastructures [7] offer optimal support for load balancing. Essentially, the infrastructure maintains the available resources of a grid and assigns tasks to the least loaded nodes. In addition, it is even possible to install new services in the grid in case a bottleneck is detected. However, these systems lack a sophisticated support for combining several service calls into processes. This is because they act more like a UDDI repository (Uniform Description and Discovery Interface) [1] focusing on optimal routing of service requests.

1.2 OSIRIS at a Glance

In this paper, we describe an new architecture for distributed and decentralized process management system. OSIRIS, short for **O**pen **S**ervice **I**nfrastructure for **R**eliable and **I**ntegrated process **S**upport, combines the benefits of several existing infrastructures, namely: i.) discovery and invocation of web services from frameworks like .NET, ii.) process support and execution guarantees from workflow management systems, iii.) late binding of service calls and load balancing from grid infrastructures, and, finally, iv.) Peer-to-Peer Processes Execution (P^3E) in analogy to peer-to-peer file systems. An interesting characteristic of P^3E in OSIRIS is that process navigation costs only accumulate on nodes of the community that are directly involved in the execution of a process. Especially, there is no central component in charge with process execution. Therefore, and in contrast to a centralized process engine, P^3E bears the potential to scale well with the number of concurrent processes and the number of service providers. First experiments based on the OSIRIS system verify this behavior.

To enable P^3E, nodes require global meta information about service providers and the load of their hosts. The approach of OSIRIS is to maintain meta information in global repositories, and to distribute the information to the nodes. Thereby, replication of meta information runs completely independent of process execution, i.e., P^3E always considers only local replicas of meta information. To reduce the amount of update messages in the system, nodes subscribe only for parts of the meta information. If meta data changes, only nodes having subscribed for this piece of information receive an update (*publish/subscribe replication*). Moreover, we relax the freshness of local replicas. For instance, nodes require load information to balance the load of service requests among all available providers. Since load information is highly dynamic, consistent replication would be very expensive. Consequently, OSIRIS only guarantees that the nodes' local copies of a piece of global meta information fulfill some freshness constraints (i.e., the difference to the correct load value never exceeds 10%).

The focus of this paper is on the presentation of the main concepts of OSIRIS and the verification of its high scalability capabilities by first performance evaluations. The latter very well supports our belief that the shift from centralized process management to decentralized solutions will boost large scale information systems. Especially, P^3E has the potential to scale well with the number of concurrent processes and the number of nodes connected to the community.

1.3 Organization

The remainder of this paper describes our approach in more detail. Section 2 and 3 describe the basic concepts and the implementation of OSIRIS, respectively. In section 4, we present fist results of a performance evaluation on the basis of the OSIRIS system. Section 5 summarizes related work, and Section 6 concludes.

2 Peer–to–peer Process Execution

In this section, we first shortly describe the *hyperdatabase* concept which is the basis for the infrastructure we have built. Then, we present the Osiris system as an implementation of a hyperdatabase in more detail.

2.1 The Hyperdatabase Concept

In short, a hyperdatabase (HDB) [17, 16, 18] provides transactional guarantees for processes over distributed components using existing services. The HDB provides sophisticated routing strategies to dynamically choose among the available providers. And finally, a hyperdatabase does not follow a monolithic system architecture but consists of an additional thin software layer, a so-called *hyperdatabase layer* (HDB layer). The HDB layer resides on every node of the community and extends existing layers like the TCP/IP stack with process related functionality. In terms of routing of requests, a service is called by only specifying its type. The HDB layer then maps the service type to a real network address taking the list of available service providers and the current load of their hosts into account. As such, the HDB layer abstracts from service routing much like TCP/IP abstracts from data packet routing. Moreover, while the TCP/IP protocol guarantees correct transfer of bytes, the HDB layer guarantees the correct shipment of process instances.

Of course, a hyperdatabase requires that each service provider locally installs an additional software layer. Ideally, this layer comes together with the operating system much like the TCP/IP stack does (comparable to the .NET framework). For the time being, the community of service providers is split into two parts: on the hand side, there are providers that cooperate and have the HDB layer installed on their hosts. On the other hand side, we can incorporate external providers via proxy components. A proxy runs the HDB layer and forwards all service invocations to an appropriate external provider. Hence, providers can join the community even without installing the HDB layer.

2.2 Architecture of Osiris

The architecture of Osiris consists of two parts: firstly, each node runs an HDB layer that is responsible for the execution of processes, routing of service requests, and failure handling. The main emphasis of our design was to avoid any central component for process navigation. Rather, the execution of a process instance involves only nodes that provide a service for that process (see the process flow in Figure 1). To do so, each HDB layer requires global meta information, e.g., about service providers and their current load. This leads to the second part of our architecture: additional global repositories maintain the global meta information about the nodes in the community (cf. middle box in Figure 1). Each HDB layer contains replica of those pieces of meta information it needs to fulfill its tasks.

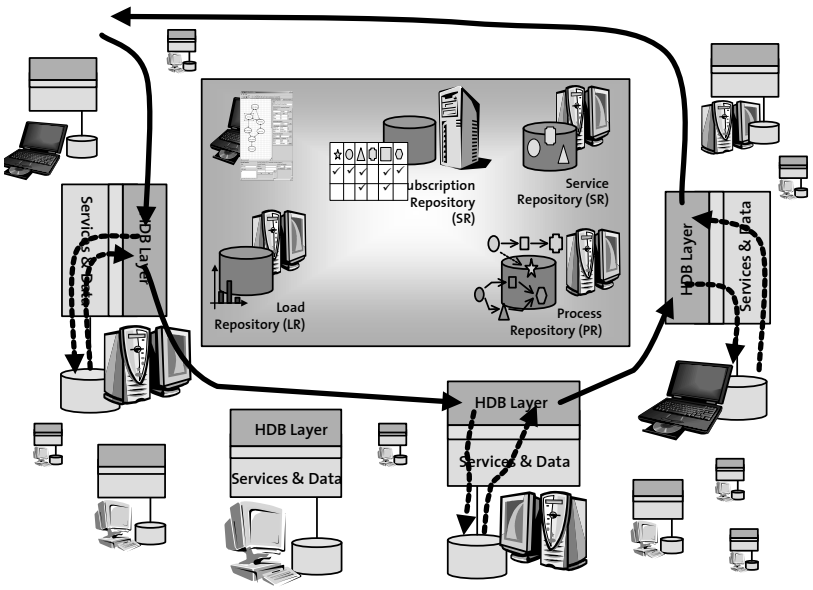


Fig. 1. Peer–to–peer process execution

It is important to distinguish between the task of process execution and the tasks of meta data replication. Process execution in Osiris follows a true peer-to-peer approach touching only nodes that provide a service for the process, and accessing meta information only locally. Meta data replication, on the other hand, is based on a hierarchical organization with central repositories (but distributed over a set of nodes), and clients (= HDB layers) replicating from them. Process execution and meta data replication run independent of each other.

Subscription Information. Usually, several providers offer semantically equivalent services. To simplify the discovery of services, Osiris deploys a publish and subscribe (pub/sub) mechanism for service invocations: a service provider subscribes for the *execution of its services* at a global service repository (similar to a UDDI repository). If a client requests a service, it publishes this request with the service type as the topic, and the HDB layer selects and invokes one of the available services. In Osiris, this means that a process instance is migrated by publishing the instance data with the service topics of subsequent steps. Of course, there is no central pub/sub component routing publications. Rather, each HDB layer holds local replicas of the global subscription lists and migrates process instances in a peer-to-peer way, i.e., plays the role of a pub/sub broker .

Required Process Information. A service invocation may lead to two result states: success or failure. In case of a success, the current execution path is followed. Otherwise, an alternative path is executed, possibly compensating some of the previous activities (partial roll back). Note that alternatives and compensation are implemented as additional execution paths in the process.

With P^{3}E, an HDB layer requires only minimal information about the process definitions, i.e., how to call the local service and whom to publish the instance data to depending on the result state.

Join Node. During the execution of process instances, the termination of a process activity might trigger an arbitrary number of succeeding activities. This is usually termed as a fork of the process execution. In a distributed environment like OSIRIS, this enables a true intra-parallel execution of a process instance. A fork is usually accompanied by a join, i.e., a step where a number of execution paths are merged. A join requires that the nodes along all the merged execution paths know *where* to join with *which* paths. To this end, OSIRIS assigns a unique join node to a process instance at its creation time. Whenever a join is required, the execution paths meet at this node before they continue their path together. Since a join in OSIRIS is modeled as an additional activity call, a join typically will not occur at a provider node. Moreover, OSIRIS can have an arbitrary number of different dedicated join nodes, i.e., different process instances have (possibly) different join nodes.

Load Information. Typically, a service is offered by different providers possibly with conditions (e.g., service costs) that may change over time. Therefore, the HDB layer selects services at runtime to optimize the throughput and response times of processes. As a consequence, the HDB layers require load information about nodes offering those service types that potentially may receive process instance data from this HDB layer.

2.3 Replication Management

So far, we did not yet address how global meta information is distributed among the nodes of the OSIRIS system. Obviously, we have to avoid that nodes request information from a central repository each time the information is needed. On the other hand, full and consistent replication may be too costly, e.g., the distribution of load information in a peer-to-peer manner would lead to $O(n^2)$ messages for n components. Our solution exploits the following characteristics:

- Updates on certain pieces of information are infrequent, e.g., process definitions or subscription lists of service providers.
- Nodes only require parts of the global information, e.g., only information about processes that may run at the node.
- Changes on the global information are not always critical, e.g., if the load of a node slightly changes, its new load does not need to be published.

Pub/Sub Replication. The basic idea of our replication scheme is based on publish/subscribe techniques: the primary copy of the data resides at a single node, i.e., a global repository. OSIRIS stores such information in a semi-structured representation (an XML document). Then, each client that wants to replicate data first has to subscribe for the document. As a result of this subscription, the repository publishes the current state of the XML document. Whenever the primary copy changes, the repository publishes corresponding update messages

to all subscribed clients. Such an update message comprises only the minimal information needed to update the clients versions to the current state.

Partial Replication. Replicating the entire XML document is not optimal if clients only require small pieces of the document. For instance, the HDB layer does not need the entire process definition to execute the current step of an instance. Another example comprises subscription information and load information: consider a large process with a service B immediately following a service A along one of the execution paths. A node running service A only must know about providers and load data of nodes running service B, but not about those offering other services for that process. In general, the HDB layer only requires a small subset of the global information. Hence, instead of subscribing for the entire XML document, a node subscribes only for a portion of it by passing an XPath expression with the subscription.

Freshness Predicates. Replication often demands that the primary copy and all replicas are consistent, i.e., they contain the same data. For some global information, this requirement is vital (e.g., process definitions). For other pieces of global meta information like subscription lists and load information, this requirement is far too strong. Consider for instance subscription lists: as long as the replicated version is *sufficiently* close to the primary copy, the HDB layer is still able to route process instances appropriately. Another example is load information: if the load of a service only marginally changes, we do not have to propagate these updates to all clients. Our generic approach allows to add a so-called *freshness predicates* to the subscription. These predicates define under what circumstances changes on the primary copy have to be published.

3 The Prototype System

In addition to the two components of hyperdatabases for process support, i.e., the hyperdatabase layer that is installed on every node of the community and core services that globally provide basic functionality required for the process management, the OSIRIS prototype also consists of O'GRAPE (OSIRIS GRAphical Process Editor) [22], an additional tool to graphically model and define processes. OSIRIS has been implemented over the past two years and is currently being applied to support process executions in the context of a virtual campus and a large scale image retrieval engine. In order to give a rough impression of the complexity of OSIRIS, the basic infrastructure together with services for the two applications consists of about 300'000 lines of code.

3.1 The Hyperdatabase layer

The local hyperdatabase layer. consists of number of plug-able modules, i.e., we can adapt the system to the needs of the application. For instance, if concurrency control is not important (non-critical application or no conflicts between service invocations), we may omit the module from the local software layer. In the following, we concentrate on the description of the most interesting modules:

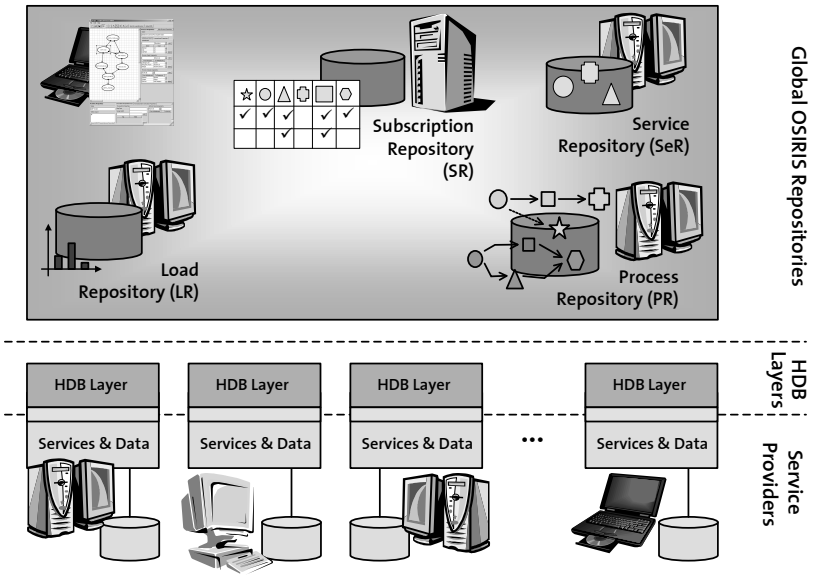


Fig. 2. OSIRIS architecture overview

- **Replication Manager.** This module provides the basic replication services as described in Section 2.3. The replication manager may take the roles of both server and client, i.e., it replicates information from remote sources and it maintains information for remote clients.
- **Process Manager.** Whenever a process instance enters the HDB layer, the process manager requests the corresponding process definition from the replication manager, and calls the specified service locally. After a service call, the process manager determines the services to be called next. For each service, it migrates the process instance to a suitable service provider for subsequent steps via the communication module. Since process definition is replicated at HDB layer, a process instance consists of a reference to the process definition, the name of the current activity, and all global process instance data.
- **Communication.** The communication module resolves pub/sub addressing and deploys the load balancing module to determine an optimal provider among the list of subscribed nodes. Communication is message-based and asynchronous. The transfer of a message between two nodes is based on a 2PC protocol over persistent queues[3]. Since the number of messages is typically quite large, we use priority queues to boost important messages.
- **Publish Subscribe Routing.** The implementation of the pub/sub module is rather simple due to the deployment of the replication manager. The

[3] Note that there are only two participants in this distributed transaction. Hence, the protocol is rather simple.

pub/sub module replicates subscription lists from the global Subscription Repository. To resolve a pub/sub address, it simply returns the list corresponding to the topic.

- **Load Balancing.** In our implementation, the load balancer chooses the provider that is able to work off the next execution unit at the earliest point in time[4]. For that purpose, the Load Balancing module replicates data from the global Load Repository via the Replication Manager. Since load information is not so critical, we use less strict freshness predicates as in the case of the pub/sub module and the Process Manager.
- **Service Manager.** The service module builds a bridge between the OSIRIS system and the local service providers. At startup time, the service manager registers all local services at the global Subscription Repository. If needed, the service manager may download new releases of service software from the Service Repository. Furthermore, the service manager keeps track of the load of all local services and publishes this information to global load repositories.
- **Concurrency Control.** The task of this module is to provide functionality for globally synchronizing process instances. This is needed in cases where concurrent process instances access shared resources and if there is a flow of information between these instances. Unlike many workflow management systems where this task is delegated to the applications, OSIRIS makes concurrency control transparent to the processes.

3.2 Core Services

The core services of OSIRIS are essential to run process in a peer-to-peer manner. Four core services maintain global repositories from which the HDB layer replicate. A further service is dedicated to concurrency control. In addition to these core services, our current prototype runs more than 100 application-specific services, mainly to support image retrieval and the virtual campus application.

- **Subscription Repository (SR)** manages a list of all services offered by the providers in the OSIRIS–community. This service is comparable to a name server in a common TCP/IP network infrastructure. Note that, in contrast to related pub/sub approaches, the subscription repository provides no routing functionality. Rather, each local component routes process instances in a peer-to-peer manner to the targeting system.
- **Service Repository (SER)** holds the interface definitions for all services available in the system. In addition, it holds executables (software) for a subset of services to support on demand installation of hot-spot services. If the load of a service type becomes large, OSIRIS can decide to install additional instances of this service type to enhance throughput.

[4] It is simple to extend this notion with more complex costs functions. For that purpose, we can exchange the load balancing module with a more sophisticated one.

- **Process Repository (PR)** holds the global definitions of all processes of the community. This service decomposes process definition to *execution units* [5] consisting of pairs of subsequent activities.
- **Load Repository (LR)** organizes and manages the load of service providers in the system. Significant changes in queue status of providers are propagated to corresponding components.
- **Concurrency Control (CC)** When a request for the invocation of a service is sent to the local HDB layer, it is forwarded to a global concurrency control service. There, correct synchronization is guaranteed by applying protocols at the process level [19]. However, an important optimization is done at the local HDB layer: before forwarding a request to the global CC service, it is first checked whether this invocation may have side-effects on other service invocations (i.e., whether potential conflicts exist). This dramatically reduces the overhead of concurrency control as it is commonly observed that conflicts at the level of service invocations are rather rare.

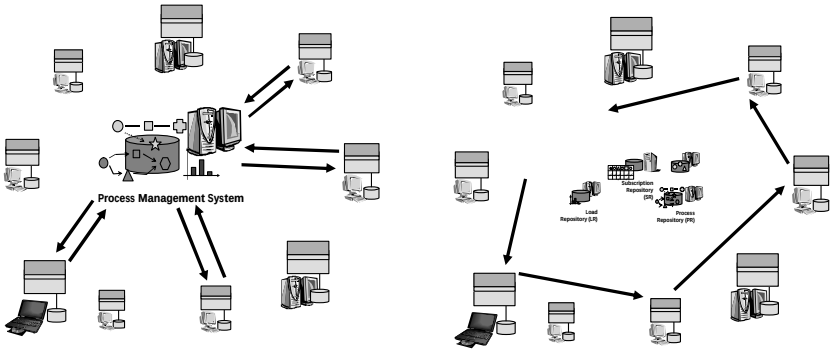
Note that the repositories do not require a centralized implementation. For instance, OSIRIS can contain a large number of process repositories. These repositories may exchange and coordinate their contents, and may serve as a repository for a limited number of nodes. If the number of nodes increase, we simply can add more repositories. To decrease the load on the other repositories, we may partition the communities into disjunctive sets of nodes. For each partition, a dedicated subscription, service and load repository is available. From time to time, these repositories exchange information and publish changes of remote partitions to the members of their own partition.

3.3 Failure Handling

OSIRIS handles failures at various levels. Firstly, *network failures* might occur when a node of the community is currently not available. If the address was determined from a subscription list, we choose an alternative address from that list (via load balancing) and continue with invoking the new service instance. If no alternatives exist or all alternatives fail, we suspend the message until new entries for the list arrive from the replication module.

Software failures denote that a service or the local HDB layer fails. If a service crashes, the local HDB layer immediately restarts the failed service and re-submits the requests the service was handling at the point in time of its failure (we do so with persistent queues). If the local HDB layer crashes (or the host on which it is running), the user (a watchdog or a reboot) must restart the software. OSIRIS then recovers by consulting the content of the local persistent queue and logging tables, subscribes all local services at the subscription repository, and re-starts the processing of the queue entries from their last persistent

[5] An execution unit is a part of a global process definition that holds all information needed to handle activity data flow, call a local service. In addition, preprocessed information about next activities are needed to determine the subsequent service calls.



(a) Central Setup (b) Distributed Setup (Osiris)

Fig. 3. System configuration settings in performance evaluations

state. Moreover, the HDB layer must check the state of message transfers at time of the crash. Depending on the log information from the 2PC protocol, messages are re-sent or locally removed.

Finally, Osiris also deals with *application failures*, i.e., a service request results in an error. However, Osiris can not handle this case of failures automatically. Rather, the application developer must resolve such failures at process definition time. For that purpose, our process model provides alternative paths that are followed if a service fails, and compensation steps and roll back steps which are executed to undo former side effects of a service invocation. Essentially, the application developer is responsible for a correct failure handling at the application level. If correctly done, Osiris guarantees that a process will terminate in a well defined state even in case of lower-level failures.

4 Measurements

In this section, we provide the results of first performance evaluations based on the Osiris system. The goal of these evaluations is to verify the high potential of the P^3E paradigm compared to a traditional centralized process engine, especially in terms of scalability.

4.1 Measurements Setup

We consider process execution on top of a set of distributed services. These services are installed on several provider nodes. The goal is to set up a simple configuration in order to measure the scalability of the navigation concept of Osiris. Figure 3(a) shows the centralized configuration. A dedicated process management node calls services of distributed service providers in a traditional request/reply style. This system configuration represents a state-of-the art process management system executing (web) service-based processes. Both process

navigation and meta data management takes place at a central node. In contrast to this, Figure 3(b) shows a typical setup of Osiris' peer-to-peer-based process execution. Process instances are navigated and routed directly at the provider nodes. Meta data is managed by a set of global services, which are not bound to a single node and which operate in an asynchronous and decoupled way.

On the basis of these two configuration settings, we want to show the impact of distributing navigation costs. To clearly factor out the effects, we use for both settings exactly the same process definitions and the same transport infrastructure. To achieve this, we have additionally implemented a central coordinator which meets these constraints.

4.2 Preliminary Results

In the following, we illustrate preliminary experiments to measure the pure navigation costs of process instances. For that purpose, we have used a process (consisting of three activities, i.e., three different service invocations) with no actual payload, i.e., the service calls just return after invocation. This way, it possible to only measure the overhead of process management. In addition to the centralized and distributed execution scheme, our experiments further consider two different service binding paradigms. In the first setting (denoted as *3 fix nodes*), the process engine does not use late service binding load balancing, i.e., services are hard-coded into the process definitions much like in a .NET application. In a second setting (denoted as *20 Nodes*), late binding and load balancing are available and the complete cluster of service providers is used.

Figure 4.2 illustrate first results with varying numbers of concurrent processes. In each experiment, we started a number (from 200 to 10,000) of processes at the start time and measured the total time to work off all process instances. This time divided by the number of processes approximates the mean overhead of process navigation for a single process instance. With a centralized process engine, there is no big difference whether late service binding and load balancing are turned on or off (upper two graphs in the plot). The main reason for this behavior is the fact that the central engine is not able to work off processes fast enough and to route them to the service providers. As a result, most processes are "endlessly" delayed in the engine's internal queue for navigation (although more than 10 threads were concurrently handling entries in the queue). CPU monitoring of the involved nodes exhibited that the coordinator node was fully loaded (CPU load at 100%) while the nodes of the service providers were idle over long phases of the run.

Decentralized process management, on the other hand, significantly improves process throughput. Even more interestingly, average navigation costs remain constant over the entire range of the number of concurrent process instances. With late binding and load balancing enabled, distributed process management always leads to navigation costs of around 50 ms per process regardless of the number of concurrent processes (from 200 up to 10,000). Without late binding and load balancing, our distributed approach also suffers from extensively long queues at the three service providers utilized by the process (with our approach

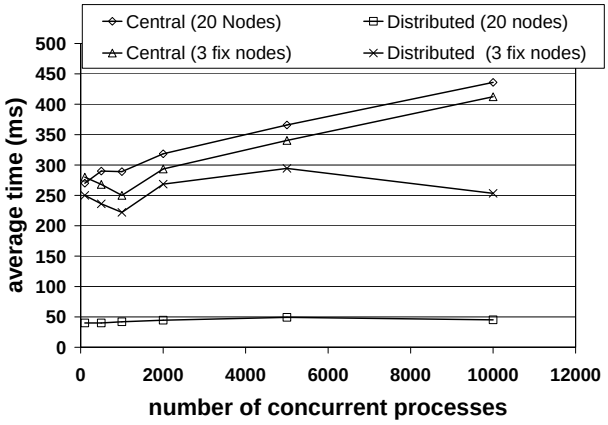


Fig. 4. Overall Process Navigation Time

in OSIRIS, they have to execute all navigation steps). But in contrast to the centralized approach, these three service providers are able to work off the process instance much faster than the central coordinator.

Based on these first experiments, we conclude that process navigation is much cheaper in a distributed environment and that a distributed process management system scales much better to larger workloads than a central coordinator. However, the advantage of decentralized process execution only pays off in combination with late binding and load balancing of service invocations. In order to further investigate the different performance characteristics of centralized and decentralized process execution, we are currently defining a benchmark for process management system.

5 Related Work

An important feature of the Osiris system is that process execution takes place in a peer-to-peer way without involving a centralized component. This is in sharp contrast to state-of-the-art process support system like MQSeries Workflow [13], BizTalk [12], etc. Although these systems allow process instances to be shipped between different distributed process engines, services within processes are nevertheless invoked from one single engine at a time in a request/response style. Yet, the peer-to-peer execution of Osiris requires that a middleware layer is installed on every participating component resp. service provider. This is very similar to frameworks like .NET [14] or CORBA [3]. However, the local middleware layers within these frameworks only support the invocation of single services without the possibility of combining services to higher level applications.

One crucial requirement for the peer-to-peer execution is that meta information has to be distributed between the local middleware layers with dedi-

cated freshness guarantees. Pure peer–to–peer file sharing approaches like FastTrack [6] or Gnutella [9] use (optimized) flooding algorithms to distribute meta data within the complete system. OSIRIS, in contrast, applies publish/subscribe techniques. Essentially, this allows for a fine-grained treatment of different meta data and to apply individual freshness guarantees.

Self-adaptability to a dynamically changing environment is another core aspect of the OSIRIS system. Essentially, this means that prior to each service invocation, a decision is needed to determine the provider of an actual service invocation. Several process management systems use publish/subscribe techniques for this purpose (Dayal et al. [4] provide a detailed overview of these systems). However, conventional implementations of publish/subscribe techniques either require a centralized publish/subscribe broker or use broadcast technologies. In contrast, OSIRIS uses neither of them. This is possible since each local hyperdatabase layer is equipped with a publish/subscribe broker that is able to correctly handle local events and to route service invocations. While service discovery is shielded in OSIRIS by the clever replication of meta information, other systems like eFlow [2] or CrossFlow [10] have to explicitly provide support for service discovery. In terms of self-adaptability, OSIRIS reacts on changes of the overall configuration (e.g., new service providers) by implicit feedback loops [23]. In addition, following the idea of service grids [7], active changes of the overall configuration are also possible by the installation of new service instances. In medical information systems, even instances of long-running patient treatment processes have to be continuously migrated to the most recent process description so as to provide up-to-date treatment knowledge. Therefore, systems like ADEPT$_{flex}$ [15] have to deal with instance evolution. Such mechanisms are however orthogonal to OSIRIS and could be seamlessly integrated to additionally enable this kind of dynamic changes.

6 Conclusions and Future Work

In order to provide an infrastructure for processes execution that is, at the same time, flexible, highly scalable, and that is able to dynamically adapt to changing environments, OSIRIS combines several concepts. First, this includes support for the discovery and invocation of web services by using established protocols like SOAP and WSDL. Second, several service calls can be combined to processes. To this end, OSIRIS is extended by O'GRAPE, a graphical modeling tool for processes. Moreover, OSIRIS provides dedicated transactional execution guarantetes for process instances. Third, flexibility and dynamic adaptation is realized by supporting late binding strategies for service invocations. This allows to dynamically chose among the available service providers and to apply sophisticated load balancing techniques. Finally, by applying clever replication algorithms for meta data, OSIRIS supports executions of process instances in a peer-to-peer style. Due to this peer-to-peer execution, a centralized process engine is avoided. As a consequence, OSIRIS is optimized for achieving a high degree of scalability, both in terms of the number of process instances in the system an in terms of

the number of service providers. First evaluations presented in this paper have shown promising results.

In order to extend these first performance evaluations, we have recently started with the specification of a fair benchmark (similar to the one presented in [8]) that allows to compare the decentralized OSIRIS prototype with a rather conventional centralized process engine. Such a possible benchmark setup consists of large number of cluster nodes acting as service providers (to this end, the 128 node Xibalba cluster of ETH Zürich will be exploited). These providers offer a different set of services. In here, it is an important task to find a balance between services that are location-dependent (i.e., that are only available with dedicated providers) or location independent services which can be installed at any node in case of a bottleneck. Hence, a benchmark application should be a well-balanced mix of data-centric services accessing, for instance, database repositories and computationally intensive services. The latter are usually location independent and can be replicated on demand. In addition to basic benchmark specifications like numbers and types of processes, this will also contain a specification on the fluctuation and dynamic behavior of available service providers in the overall system. Evaluations based on this benchmark are performed within a joint project –together with the IBM labs– that has recently been settled.

Finally, in order to provide a truly peer-to-peer execution even in the presence of services with a large number of conflicts, the current global concurrency control service will be replaced by a distributed implementation. Although concurrency control is a problem that requires a global solution, we have started to investigate a decentralized implementation that replaces synchronization at a global component with communication between the individual HDB layers [11].

Acknowledgement

We thank Frank Leymann for constructive comments and many stimulating discussions.

References

1. Ariba, IBM, and Microsoft. UDDI Technical White Paper. `http://www.uddi.org`.
2. F. Casati, S. Ilnicki, L. Jin, V. Krishnamoorthy, and M. Shan. Adaptive and Dynamic Service Composition in eFlow. In *Proc. Conf. on Advanced Information Systems Engineering*, Stockholm, 2000.
3. CORBA – Common Object Request Broker Architecture. `http://www.omg.org/`.
4. U. Dayal, M. Hsu, and R. Ladin. Business process coordination: State of the art, trends, and open issues. In *Proceedings of 27th International Conference on Very Large Data Bases, Roma, Italy*, September 2001.
5. M. Schmid F. Leymann, D. Roller. Web services and business process management. *IBM Systems Journal*, 41(2):198–211, 2002.
6. FastTrack – P2P Technology. `http://www.fasttrack.nu`.
7. I. Foster, C. Kesselmann, J. Nick, and S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. `http://www.gridforum.org/ogsi-wg/`.

8. M. Gillmann, R. Mindermann, and G. Weikum. Benchmarking and configuration of workflow management systems. In *Cooperative Information Systems, 7th International Conference, CoopIS 2000, Eilat, Israel, September 6-8, 2000, Proceedings*, pages 186–197, 2000.
9. Gnutella RFC. `http://rfc-gnutella.sourceforge.net`.
10. P. Grefen, K. Aberer, H. Ludwig, and Y. Hoffner. CrossFlow: Cross–Organizational Workflow Management for Service Outsourcing in Dynamic Virtual Enterprises. *IEEE Data Engineering Bulletin*, 24:52–57, 2001.
11. K. Haller and H. Schuldt. Consistent Process Execution in Peer-to-Peer Information Systems. In *Proceedings of the 15th Conference on Advanced Information Systems Engineering (CAiSE 2003)*, pages 289–307, Klagenfurt/Velden, Austria, 2003. Springer LNCS, Vol. 2681.
12. B. Metha, M. Levy, G. Meredith, T. Andrews, B. Beckman, J. Klein, and A. Mital. Biztalk Server 2000 Business Process Orchestration. In *IEEE Data Engineering Bulletin 24(1)*, 2001.
13. IBM MQSeries Workflow. `http://www.ibm.com/software/ts/mqseries/workflow/`.
14. Microsoft .NET. `http://www.microsoft.com/net/`.
15. M. Reichert and P. Dadam. ADEPT$_{\text{flex}}$ — Supporting Dynamic Changes of Workflows without Losing Control. *Journal of Intelligent Information Systems*, 10(2):93–129, March 1998.
16. H.-J. Schek, K. Böhm, T. Grabs, U. Röhm, H. Schuldt, and R. Weber. Hyperdatabases. In *Proceedings of the 1st International Conference on Web Information Systems Engineering (WISE'00)*, pages 14–23, Hong Kong, China, June 2000.
17. H.-J. Schek, H. Schuldt, C. Schuler, and R. Weber. Infrastructure for information spaces. In *Proceedings of Advances in Databases and Information Systems, 6th East European Conference, ADBIS 2002*, volume 2435 of *Lecture Notes in Computer Science*, pages 23–36, Bratislava, Slovakia, September 2002. Springer.
18. H.-J. Schek, H. Schuldt, and R. Weber. Hyperdatabases – Infrastructure for the Information Space. In *Proceedings of the 6th IFIP 2.6 Working Conference on Visual Database Systems (VDB'02)*, Brisbane, Australia, May 2002.
19. H. Schuldt. Process Locking: A Protocol based on Ordered Shared Locks for the Execution of Transactional Processes. In *Proceedings of the 20th ACM Symposium on Principles of Database Systems (PODS'01)*, pages 289–300, Santa Barbara, California, USA, May 2001. ACM Press.
20. H. Schuldt, G. Alonso, C. Beeri, and H.-J. Schek. Atomicity and Isolation for Transactional Processes. *ACM TODS*, 27(1), March 2002.
21. SOAP – Simple Object Access Protocol. `http://www.w3.org/TR/SOAP/`.
22. R. Weber, C. Schuler, H. Schuldt, H.-J. Schek, and P. Neukomm. WebService Composition with O'GRAPE and OSIRIS. In *Proc. of 29rd International Conference on Very Large Data Bases*, Berlin, Germany, September 2003.
23. G. Weikum, A. Mönkeberg, C. Hasse, and P. Zabback. Self-tuning Database Technology and Information Services: from Wishful Thinking to Viable Engineering. In *PProceedings of 28th International Conference on Very Large Data Bases*, pages 20–31, Hong Kong, China, August 2002.
24. G. Wiederhold, P. Wegner, and S. Ceri. Towards Megaprogramming. *Communications of the ACM*, 35(11):89–99, November 1992.
25. I. Wladawsky-Berger. Advancing the Internet into the Future. Talk at the *International Conference Shaping the Information Society in Europe 2002*, April 2002. `http://www.ibm.com/de/entwicklung/academia/index.html`.
26. WSDL – Web Service Description Language. `http://www.w3.org/TR/wsdl/`.